Západočeská univerzita v Plzni

Fakulta aplikovaných věd

DISERTAČNÍ PRÁCE

2014

Ing. Pavel Bžoch

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

ZVYŠOVÁNÍ VÝKONU A UDRŽOVÁNÍ KONZISTENTNOSTI DAT V MOBILNÍCH ZAŘÍZENÍCH PRO DISTRIBUOVANÉ SYSTÉMY SOUBORŮ

Ing. Pavel Bžoch

disertační práce

k získání akademického titulu doktor

v oboru Informatika a výpočetní technika

Školitel: Prof. Ing. Jiří Šafařík, CSc.

Katedra: Informatiky a výpočetní techniky

Plzeň 2014

University of West Bohemia
Faculty of Applied Sciences

INCREASING PERFORMANCE AND MAINTAINING DATA CONSISTENCY IN MOBILE DEVICES FOR DISTRIBUTED FILE SYSTEMS

Ing. Pavel Bžoch

Doctoral Thesis

in partial fulfilment of the requirements for the degree of

Doctor of Philosophy in specialization

Computer Science and Engineering

Supervisor: Professor Jiří Šafařík

Department of Computer Sciences and Engineering

Czech Republic, Pilsen 2014

Prohlášení

Předkládám tímto k posouzení a obhajobě disertační práci zpracovanou za závěr doktorského studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji, že jsem tuto práci vypracoval samostatně s použitím odborné literatury a dostupných pramenů uvedených v seznamu, který je součástí této práce.

Declaration

I do hereby declare that this doctoral thesis is a result of my own work and that it has been solely completed by myself. Where appropriate, I have made acknowledgement of others' work.

V Plzni, 08.08.2014

Ing. Pavel Bžoch

Abstract

Mobile devices such as mobile phones, notebooks, netbooks, PDAs are spread worldwide. The computational and storage capacity of these devices has grown recently especially in mobile phones branch. By using a cellular network or Wi-Fi, these devices can also access internet and operate with remotely stored files. For increasing availability and reliability of accessed files, distributed file system can be used as a remote storage.

This thesis deals with access acceleration to files from mobile devices. We cannot influence the speed of the mobile network, but we can suppose that once requested file will be requested again in the future. Thus, we use an intermediate component called cache. Because the cache has limited capacity, the old cached content has to be replaced if new data need to be stored in the cache. For making a replacement decision, novel algorithms using local and server statistics are presented.

The cached file may be in inconsistent state when the original file at the server side is changed. Accessing these files is undesirable. Thus, we propose algorithms for maintaining cache consistency.

All introduced algorithms are also evaluated.

Abstrakt

Mobilní zařízení jako jsou mobilní telefony, notebooky, netbooky nebo PDA jsou celosvětově rozšířena. Výpočetní kapacita a také perzistentní paměť se u těchto zařízení stále zvyšuje. Obzvláště lze tento trend pozorovat u mobilních telefonů. Pokud použijeme celulární sítě nebo Wi-Fi, můžeme z těchto zařízení přistupovat k vzdáleně uloženým souborům. Protože uživatelé požadují po vzdáleném úložišti vysokou dostupnost a spolehlivost, je vhodné použít distribuovaný systém souborů.

Tato práce se zabývá urychlením přístupů ke vzdáleným souborům z mobilních zařízení. Z pohledu klientské aplikace není možné ovlivnit rychlost přenášení souborů, ale lze předpokládat, že jednou požadovaný soubor bude znovu požadován v budoucnosti. Pro uložení těchto souborů se používá mezipaměť (anglicky cache). Protože má mezipaměť omezenou kapacitu, musí být starý obsah nahrazen novým, pokud je potřeba. V textu práce jsou představeny původní algoritmy, které slouží pro toto rozhodování o nahrazení. Tyto algoritmy používají pro rozhodování lokální statistiky a také statistiky ze serveru. Navržené algoritmy jsou také zhodnoceny.

Soubory, které jsou uloženy v mezipaměti, mohou být v nekonzistentním stavu, pokud jsou originální soubory na serveru změněny. Přistupování k souborům v nekonzistentním stavu je nežádoucí. V práci jsou proto představeny a zhodnoceny algoritmy pro udržování konzistentní mezipaměti.

Acknowledgements

At this place I would like to thank all the people who have helped me during my Ph. D. studies. First, great thanks go to my supervisor Prof. Ing Jiří Šafařík, CSc. for his support during my research and Ph. D. thesis finalization.

I am dedicating this work to my parents, who have always supported me in both happier and tougher times. Special thanks go to Kristina, my moral support.

Great thanks go also to all my colleagues in the Department of Computer Science and Engineering who were discussing many scientific problems with me. They gave me various inspiring ideas and theories, which helped me in my research.

Table of Content

List of Tables

List of Figures

List of Abbreviations

3G	Third Generation of mobile telecommunications technology
ACL	Access Control List
AES	Advanced Encryption Standard
AFS	Andrew File System
AS	Authentication Server
CBC	Cipher Block Chaining
CBPA	Criteria Based Primary-copy Assignment
CBR	Criteria Based Replication
CBRP	Criteria Based Replica Placement
CIFS	Common Internet File System
DCS	Domain Consistence Servers
DFS	Distributed File System
DFSR	Distributed File System Replication
DHT	Distributed Hash Table
DMA	Domain Manage Agent
DS	Distributed System
EDGE	Enhanced Data for GSM Evolution
EFS	Encrypting File System
ERA	Entity Relationship Attribute
EXT	Extended Filesystem
FS	File System
FUP	Fair User Policy
FUSE	Filesystem in Userspace
GPRS	General Packet Radio Service
HDD	Hard Disk Drive
HFS	Hierarchical File System
HSDPA	High-Speed Downlink Packet Access
IA	Interface Agent
IBE	Identity Based Encryption
IBM	International Business Machines Corporation
ID	Identification/Identity/ Identifier
IO	Input/Output
IP	Internet Protocol
IR	Invalidation Report
ISO	International Organization for Standardization
KDC	Key Distribution Centre
LAN	Local Area Network
LFU	Least Frequently Used
LRU	Last Recently Used
LSM	Log-Structured Merge tree
MAFS	Multi-volume Archive File System
MAN	Metropolitan Area Network
MANET	Mobile Ad-hoc Networks
MDS	Metadata Storage
MMA	Main Management Agent
MSFSS	Mass Small Files Storage System
NAT	Network Address Translation
NFS	Network File System
NTFS	New Technology File System
OPT	Optimal
OS	Operating System
OSI	Open Systems Interconnection
P2P	Peer to Peer

PC	Personal Computer
PDA	Personal Digital Assistants
QoS	Quality of Services
RAID	Redundant Array of Inexpensive/Independent Disks
RAM	Random-Access Memory
RDC	Remote Differential Compression
RMI	Remote Method Invocation
RPC	Remote Procedure Call
RW	Read Write
SHA	Secure Hash Algorithm
SMB	Server Message Block
SSL	Secure Sockets Layer
ST	Service Ticket
TCP	Transmission Control Protocol
TGS	Ticket Granting Server
TGT	Ticket Granting Ticket
TLS	Transport Layer Security
TTL	Time-To-Live
UDP	User Datagram Protocol
UIR	Updated IR
UML	Unified Modelling Language
UPS	Uninterruptible Power Supply
VFS	Virtual File System
WA	Working Agent
WAN	Wide Area Network
Wi-Fi	Wireless Fidelity
XFS	Extended File System

1 Introduction

The need of storing huge amounts of data has grown over the past years significantly. Whether data are of multimedia types (e.g. images, audio, or video) or are produced by scientific computation, they should be stored for future reuse or for sharing among users. Users also need their data as fast as possible. Data files can be stored on a local file system, on remote file system or on a distributed file system.

A local file system provides the data quickly but does not have enough capacity for storing a huge amount of the data. The data can be accessed only locally; a remote access is usually not possible. A local storage also does not provide high level of availability.

A remote file system in comparison to local file system can provide data for remote users. Other attributes are the same as above.

A distributed file system (DFS) is a part of distributed system (DS) and provides many advantages such as reliability, scalability, security, capacity, etc. In the next text, we focus on distributed systems.

1.1 Distributed Systems

Modern computations require powerful hardware. One way of gaining results faster is getting new hardware over and over again. Buying a supercomputer is, however, not a cheap solution. It also takes plenty of time to install software to these new supercomputers. Another way in achieving better system performance is using a distributed system. In a distributed system, several computers are connected together usually by a computer network. Now, we can characterize distributed system with a simple definition:

> *A distributed system is a collection of independent computers (nodes) that appears to its users as a single coherent system.* **[1]**

This concept brings many advantages. Better performance can be achieved by adding new computers to the existing system. If any of the computers crashes, the functionality of the system is still available. Using DS brings several problems

too. In the distributed systems, we have to solve synchronization between computers, data consistency, fault tolerance etc. There are many algorithms which solve these problems. Some of them are described in [**1**]. After this simple introduction, we continue with definition of distributed file systems.

1.2 Distributed File Systems

DFS as a part of DS do not directly serve to data processing. They allow users to store and share data. They also allow users to work with these data as simply as if data were stored on the user's own computer.

Compared to a traditional client-server solution, where data are stored on one server, important or frequently required data in DFS can be stored on several nodes (node means a computer operating in a DFS). This is called *replication*. Replication can be used for achieving high *reliability* of the system in comparison to systems where the replication is not used.

Data stored in DFS which uses replication are more protected in a case of a node failure. If one or more nodes fail, other nodes are able to provide all functionality. This property is also known as *availability* or *reliability.* The difference between availability and reliability is simple. Availability means that the system can serve client a request at a moment when the client connects to the system. Reliability means that the system is available all the time when the client is connected to it.

Files can also be moved among nodes. This is typically invoked by an administrator and it is done for improving a load-balancing among nodes. The users should be unaware of where the services are located and also the transferring from a local machine to a remote one should be transparent [**2**]. In DFS, this property is known as *transparency.*

If the capacity of the nodes is not enough for storing files, new nodes can be added to the existing DFS to increase DFS capacity. This property is known as *scalability*.

A client usually communicates with the DFS using a computer network, which is not a secure environment. Clients must prove their identity, which can be

done by authenticating themselves to an authentication entity in the system. The data which flow between the client and the node must be resistant against attackers. This property is known as *security*.

1.3 Aims of the Thesis

The data from a remote storage can be accessed from personal computers (PCs). A personal computer is usually connected to the server by using a wired connection. This connection has usually constant speed and the data can be accessed any time.

Nowadays, the mobile devices are commonly used. Under the term "mobile device", we can include cell phones, personal digital assistants (PDA), smart phones, netbooks, tablets etc. Also the performance and the storage capacity of these devices have been enhanced. Smart phones can have up to 4-core processors like personal computers. For storing users' data, microSD cards are usually used. The capacity of microSD card can be up to 128GB [**3**]. Having this potential, the mobile devices, namely cellular phones, can be used not only for making calls and sending SMSs, but also for creating and playing multimedia files, accessing the internet or reading and writing e-mails or accessing files from remote storage.

Multimedia files, the results of distributed computing and also binaries of mobile applications demand lots of storage space. Though the capacity of commonly used microSD cards is increasing, they are still not sufficient to store all the demanded data. In this case, the data can be stored on a remote storage and accessed via wireless network connection. When remote storage is chosen to store users' content, users wish to access data as simply as if data were stored locally. Additionally, other demands such as reliability, availability or security are required from the remote storage. Higher level of security can be achieved by distributing the security issues to the nodes participating in DFS. Use of a distributed file system can satisfy all these requirements.

The wireless connectivity is the biggest disadvantage of accessing the data remotely. Wireless technologies can provide very slow connectivity and can be expensive (e.g. cellular networks with old technologies) or can be quick and relatively cheap such as Wi-Fi. Nowadays, new technologies for increasing network

speed are being put into service (e.g. LTE [**4**]). However, these technologies are not available everywhere. They are usually put into service in big cities first and only then expand to rural areas. The price of the data transfer is still high. Moreover, time division multiple accesses with preference for voice is used to bundle the data transfer and voice service. Thus, the speed of the data transfer is reduced when there are many phone calls [**5**].

The aim of this thesis is to develop new caching policies whose use in a cache algorithm increases efficiency of data access from mobile devices. Use of a cache can also reduce the network communication. If the client applications use the cache, the problem with consistency of cached files can arise. Thus, the second goal of this thesis is a design of algorithms which serve for consistency control. All algorithms should be also evaluated and compared to existing ones. This is the third goal of this thesis.

1.4 Structure of the Thesis

The structure of the thesis is as follows. Chapter 2 covers the state of the art in DFS. It covers description of traditional DFS. Then, the algorithms for increasing performance, reliability and security are described.

Chapter 3 provides KIV-DFS architecture. KIV-DFS is an experimental DFS which is being developed at the Department of Computer Science and Engineering (Katedra Informatiky a Výpočetní techniky), University of West Bohemia.

Chapter 4 presents new caching policies which are suitable also for mobile clients. The chapter covers description of the policies, the pseudo-code of the policies and discusses computational complexity of the algorithms.

Chapter 5 presents new consistency control algorithms. These algorithms were developed considering mobile clients. The chapter covers description and pseudo-code for the algorithms.

In chapter 6, an evaluation of the proposed algorithms using a wired client and a tool called CacheSimulator is described. This tool serves for evaluation of caching policies and consistency control algorithms. The results of the algorithms behaviour gained from CacheSimulator are introduced in this chapter.

2 State of the Art in DFS

In this chapter, we provide state-of-the-art in distributed file systems. We start with the description of traditional distributed file system. We discuss algorithms for reliability, for increasing performance and security in these systems. Then, new trends in DFS are described. All new trends in DFS were also published in [**6**], [**7**], [**8**] and [**9**].

2.1 Traditional DFS

In this subchapter, traditional distributed file systems are described. These systems are called traditional because of their frequent usage. Also, many new solutions are based on these systems. Some of the traditional DFS are commercial (like AFS), and others are free (OpenAFS, NFS, Coda or Samba). We focus on NFS4, OpenAFS, Coda and SMB. All traditional DFS were developed some time ago. They use algorithms that were commonly used at this time. Since the development of these DFSs was finished, new algorithms for increasing reliability and performance were evolved. These new trends are described in 2.5.

Before we describe traditional DFS, we focus on DFS generally. Every distributed file system consists of several nodes (a node is a computer operating in DFS). Some of these nodes serve for storing file content; others serve for storing metadata about stored data (file content). File content is usually stored on a local file system as a file. Metadata are usually stored in a database. Every database record has to have a link to the file storage to the file content.

File content and metadata are usually created during file upload process. A user sends whole file with its attributes to DFS. On the server side, file content and metadata are separated and sent to the relevant storage. Whole upload process is depicted in Fig. 2.1.

When clients want to download the file back to their computers, they contact DFS. In DFS, the metadata storage provides attributes of the file and a link to the data storage where the file content is stored. This information is then sent back to the client. The client then contacts the data storage for getting the file content. The

whole download process is depicted in Fig. 2.2. In this case, we do not take into consideration the file locking.

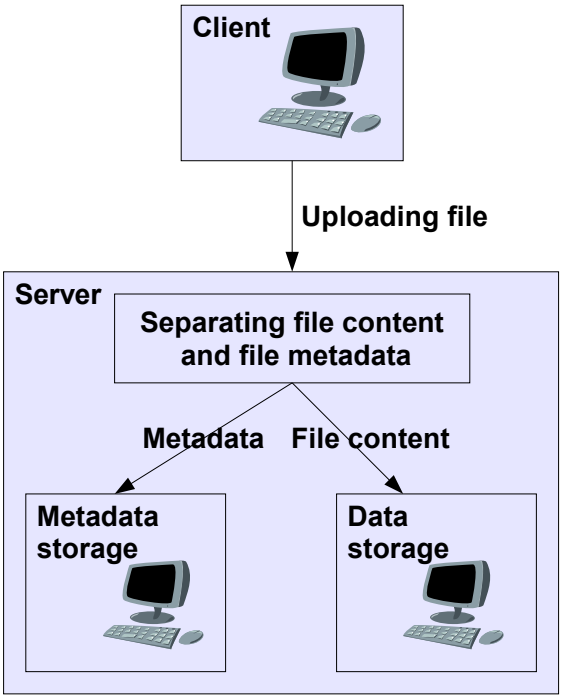

Figure 2.1: File uploads process

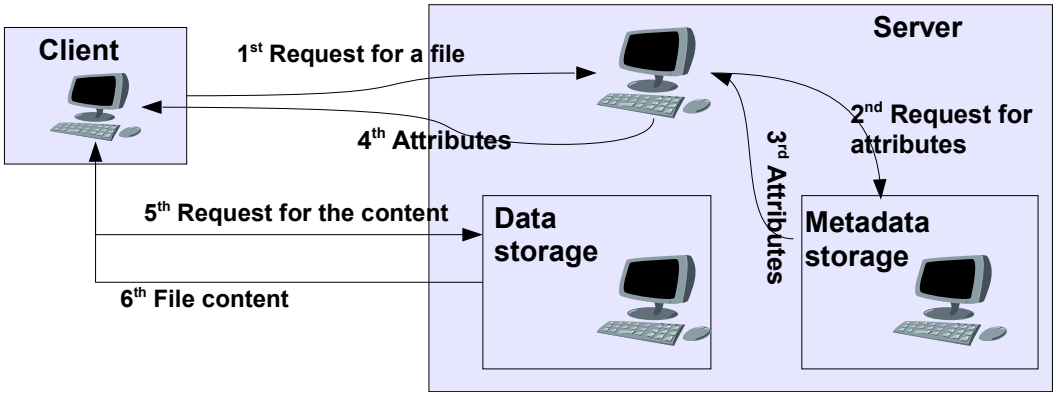

Figure 2.2: File downloads process

2.1.1 AFS

AFS (Andrew File System) was originally created at Carnegie Mellon University; later it became a commercial product supported by IBM. Now, it is being developed under a public license (OpenAFS). The design goal of AFS was to create a system for large networks **[10]**.

The smallest operating entity in AFS is a *whole file* which is the basic unit of data movement and storage. Whole file was chosen rather than some smaller unit such as physical or logical record [**11**].

AFS uses a uniform directory structure on every node. The root directory is `/afs`. This directory contains other directories which correspond to the *cells.* Cells usually represent several servers which are administratively and logically connected. One cell consists of one or more *volumes*. One volume represents a directory sub-structure, which usually belongs to one user. These volumes can be located on any AFS server. Volumes can be also moved from one AFS server to another. Moving volumes does not influence the directory structure.

Information about the whole system is stored in a special database server [**12**]. For minimizing client-server communication, AFS supports client-side caching. Cached files can be stored on a local hard disk or in a local memory. Frequently used files are permanently stored in the cache.

AFS does not provide access rights for each file stored in the system, but it provides directory rights. Each file inherits access rights from the directory where the file is located. An access list specifies various users (or groups of users) and, for each of them, specifies the class of operations they may perform [**11**].

For achieving better performance of read-only files, *snapshots* or *clones* are used. These snapshots are then stored on *replica servers.* Snapshots are usually made periodically. When the other servers are overloaded, the replica server provides files to the clients instead of these servers.

AFS uses Kerberos [**13**] as an authentication and authorization mechanism. More information about AFS can be found in [**14**]. AFS is a very stable and robust system and it is often used at universities.

2.1.2 NFS

NFS (Network File System) is an internet protocol which was originally created by Sun Microsystems in 1985, and was designed for mounting disk partitions located on remote computers.

NFS is based on RPC (Remote Procedure Call) and is supported in almost all operating systems. The NFS client and server are parts of the Linux kernel. The Kerberos system is used for user authentication. NFS was made for making client unaware of location of their files. NFS remote file system can be mounted into local directory structure. Clients can then work with their files as if the files were stored on their local file system. On server side, NFS uses the same directory structure as is shown after mounting to the client. Currently, NFS exists in several versions. In the next text, we focus on the latest version NFSv4.

In NFS, there usually exists an *automounter* on client side [**15**]. An automounter is a daemon which automatically mounts and unmounts NFS file system as needed. It also provides ability to mount another file partition if the primary partition is not available at a given moment. List of replicas must be made before automounter daemon is run.

In NFSv4, system performance is increased by using a local client cache. NFS can be extended into pNFS (parallel NFS), which contains one more server called metadata server. The metadata server can connect a file system from any data server to a virtual file system. It also provides information about the file location to the clients. When clients write file content, they must also ensure file updating on all servers where the file is located.

NFS communicates on one port since version 4 (previous versions used more ports), so it is easy to set up a firewall for using NFS4 [**12**]. The difficulty with NFSv4 is with clients. There are no suitable clients for all operating systems. More information about NFS can be found in [**16**].

2.1.3 Coda

Coda was developed at Carnegie Mellon University in 1990. It is based on the AFS idea and is implemented as a client and several servers. This system was mainly designed to achieve high availability.

The client uses a local cache. Cached files can be used by clients even after disconnection from Coda server. This feature is called *off-line operations*. While the client is disconnected from the server, all changes made to files are stored in a local

cache. After reconnecting to the server, all these changes are propagated to the server. If any collision occurs, the user has to solve it manually.

Coda uses Kerberos as an authentication and authorization mechanism. Servers provide file replication for achieving availability and safety. Coda uses RPC2 for communication. Servers store information about files which are in the client's cache [13]. When one of the cached files is updated, the server marks this file as non-valid.

The difference between Coda and AFS is in replication. Both of these systems use replication for achieving reliability. Coda uses optimistic replication; AFS uses pessimistic replication method. Pessimistic replication means that the replicas in AFS are read-only and present a snapshot of the system. Optimistic replication means that all replicas are writable. Client in Coda system must ensure file updating in all given replicas.

2.1.4 SMB (Samba)

All mentioned distributed file systems were originally made for Linux/UNIX systems. SMB was developed in 1985 by IBM as a protocol for sharing files and printers. In 1998, Microsoft developed a new version of SMB called Common Internet File System (CIFS), which uses TCP/IP for communication.

SMB has been ported to other operating systems where the SMB is called *samba*. This system is stable, wide-spread and comfortable. SMB can use Kerberos for authentication and authorization of users. It does not use local client-side caching. SMB uses the operating system's file access rights. SMB is wide used in Windows™ operating systems.

2.2 Reliability in Traditional DFS

Recall to the Introduction, reliability is a property which guarantees clients all functionality all the time when clients are connected to the system. Reliability in DFS can be achieved by using several techniques and methods. We can divide these methods into three categories: reliable communication, one node reliability, and reliability of the whole system.

2.2.1 Reliable Communication

For communication between the DFS and a client, a computer network is usually used. In a computer network, several protocols can be used. These protocols can be connectionless or connection-oriented. In the mostly used TCP/IP model, UDP is a connectionless protocol, and TCP is a connection-oriented protocol. For achieving reliability, DFS usually use connection-oriented protocols. Both TCP and UDP protocols are protocols on transport layer in ISO/OSI model as is depicted in Fig. 2.3. Each of these protocols uses port for differentiation of applications which the message will be delivered to. TCP and UDP provide end-to-end connection.

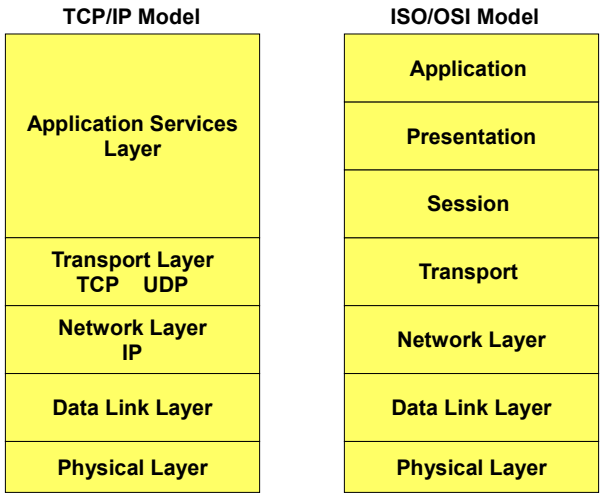

Figure 2.3: Difference between TCP/IP model and ISO/OSI model

Advantage of this solution is that the client knows almost immediately that a failure occurred and can attempt to set up a new connection. Disadvantage of using connection-oriented protocols is slowness of these protocols in comparison to connectionless protocols. Connection-oriented protocols usually have bigger overhead than connectionless protocols.

2.2.2 One Node Reliability

Once the communication is solved by using connection-oriented protocol, reliability is the next issue to be solved on the server side. Reliability on the server side in DFS can be solved by using several techniques.

Reliability of one node means that the node is still service-able while there is a fault (e.g. HDD is down). HDD crash can be prevented by using RAID. RAID is an acronym for Redundant Array of Inexpensive/Independent Disks. While using

RAID, we prevent a failure by using more hard disk. There exists several methods in storing data on RAID and there exists some kind of RAID's:

- RAID 1 (HDD mirroring). In the system, there must be two hard disks. Same data are then stored on both of these disks. This option guarantees that the node provides all functionality when one HDD fails.
- RAID 2 (bit-level striping with dedicated Hamming-code parity). This RAID requires (*N+1*) hard disks. The data are stripped by bites over *N* disks. On the last disk, parity Hamming-code is stored. This option guarantees that the RAID is protected from one hard disk failure. If data disk crashes, the data can be recovered from others and from Hamming-code. If the last disk fails, Hamming-code can be calculated from others hard disks. Parity disk is a bottle-neck of the system. If there is write request to the system, parity must be recalculated and restored.
- RAID 3 (byte-level striping with dedicated parity). RAID 3 is similar to RAID 2 but it uses bytes striping instead of bit striping.
- RAID 4 (block-level striping with dedicated parity). The data are stripped by blocks. Parity is stored on dedicated disk. Otherwise, RAID 4 is similar to RAID 2 and 3.
- RAID 5 (block-level striping with distributed parity). The data are stripped by blocks and stored on hard disks. Parity is distributed over all disks in RAID5. Distributed parity eliminates bottle neck from RAID 2-4 which have dedicated parity disk. RAID 5 needs at least three hard disks, and it is resistant to one disk failure.
- RAID 6 (block-level striping with double distributed parity). RAID 6 is similar to RAID 5, but it uses two methods in calculating parity. Parity is distributed over participated disks. RAID 6 requires at least four disks, and is protected from two disks failure.

Not all of named methods are suitable for storing data. RAID 1, RAID 5 and RAID 6 are commonly used. RAID 1 is faster than RAID 5 and 6, but it has lover capacity compared to RAID 5 and 6. RAID 5 and 6 has bigger capacity, but the parity has to be re-/calculated while writing new file content.

Preventing inconsistent state of the file system can be also done by choosing a suitable file system which will be used for storing of the data. File systems use journal techniques to prevent an inconsistent state. A journal technique or strategy describes when and for what is the journal used for.

Mostly used strategies are:

- *Writeback mode.* While using this strategy, the data blocks are directly written to the disks. Metadata are journaled. This approach prevents metadata corruption, but the data corruption can occur (if the metadata are updated before the data are written to the disk and the system crashes). [**17**]
- *Ordered mode.* To prevent an inconsistent state in the writeback mode, ordered mode writes the data block before journaling the metadata. If the system crashes during writing of data, the system can simply turn back to a consistent state. [**17**]
- *Data mode.* In this mode, both metadata and data are journaled before writing changes to the disk. This degree of protection provides the highest level of disk protection against corruption. On the other hand, this strategy is the slowest, because it must write data twice. Once the data are written to the journal and then to the disk. [**17**]

A journal technique has usually four steps:

1) The first step is writing a change of the file system into the journal.
2) The second step is applying the change to the file system.
3) The third step is writing the end of the operation into the journal.
4) The fourth step is clearing the record from the journal.

Journal techniques use e.g. EXT3, EXT4, ReiserFS or XFS in Linux/Unix systems, NTFS in Microsoft Windows systems, HFS+ in Mac OS X systems. AFS and Coda require a journal file system. Additionally, AFS client needs EXT2 file system for storing a local cache [**18**]. NFS can be run on both journal and non-journal file systems.

File system can be also damaged when the power supply is down. To prevent this state, an Uninterruptible Power Supply (UPS) can be used. UPS allows

the nodes to save all critical data to prevent file system inconsistency in a case of power shortage.

2.2.3 Whole System Reliability

Reliability of the whole system can be achieved by file replication. Replication in DFS means that the files are stored on several nodes in DFS. When one of the nodes crashes, the data are still available from the other nodes. When the system uses data replication, the original data are usually called *primary replica* or *master replica;* other copies are called *replicas*.

Replication can be done automatically or administratively. Administrative file replication means that the administrator chooses what will be replicated, and a location for replicas. Both, AFS and NFS, use administrative replication. None of the traditional DFS uses automatic replication.

In AFS and Coda, the smallest replication entity is a volume [**19**]. NFS supports file replication since version *4*, previous versions do not support replication. NFS can replicate only the whole file system [**1**].

Replication can be done in an optimistic and a pessimistic way. AFS and NFS use pessimistic way [**14**]. Coda uses optimistic way [**19**]. During a replication, all mentioned DFS use file locking. There are usually two types of locks, an exclusive lock for writing and a shared lock for reading the data.

2.3 Increasing Performance in Traditional DFS

Performance in traditional DFS can be increased by using a file replication and a caching mechanism.

2.3.1 Replication

In this sub-section, we focus on replication for achieving performance. By using replication, choosing the data and the place for replication is very important [**20**]. The data for replication should be read very often and should not be modified very often. Writing or updating a replicated data is an expensive operation. Choosing a place for a replica is also very important. The server which is chosen for storing the replica should not be over-loaded and should have good network connectivity.

For using in Windows™, Microsoft developed a Distributed File System Replication (DFSR) service. This service provides *multi-master* replication and keeps folders synchronized on multiple servers [**21**]. DFSR uses a new compression algorithm called Remote Differential Compression (RDC). RDC can be used to update replicas over a limited-bandwidth network efficiently. RDC detects removals, insertions, and rearrangements of the data in the files. Based on this information, DFSR replicates only the deltas (changes) when the files are updated [**21**].

2.3.2 Caching

A *cache* in the computer system is a component which stores data that were frequently requested, hence can be potentially used in the future. When the cached data are requested, the response time is shorter than when the data are not in a cache and must be downloaded. The cache can be stored in RAM for fast access and/or on hard disk.

A cache can be on both side of communication. On server side, the cache is usually located in RAM. On client side, the cache can be located in RAM or on hard disk. The server stores in the cache the data which are frequently requested by clients (see Figure 2.4). The client stores in the cache the data which may be requested again in the future (see Figure 2.5). Client-side caching is also sometimes called client initiated replication.

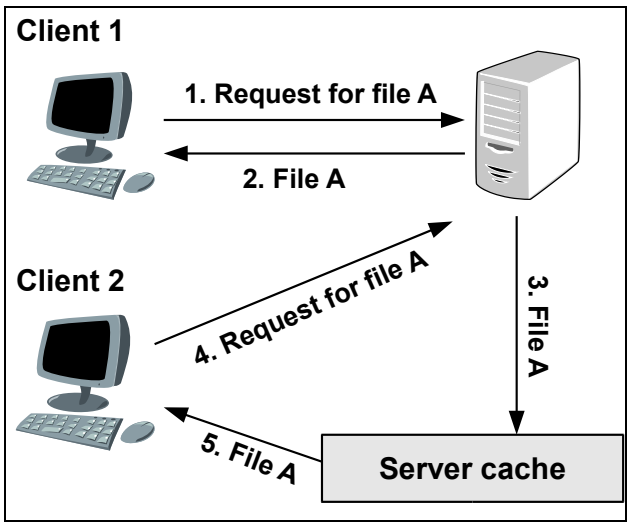

Figure 2.4: Server-side caching

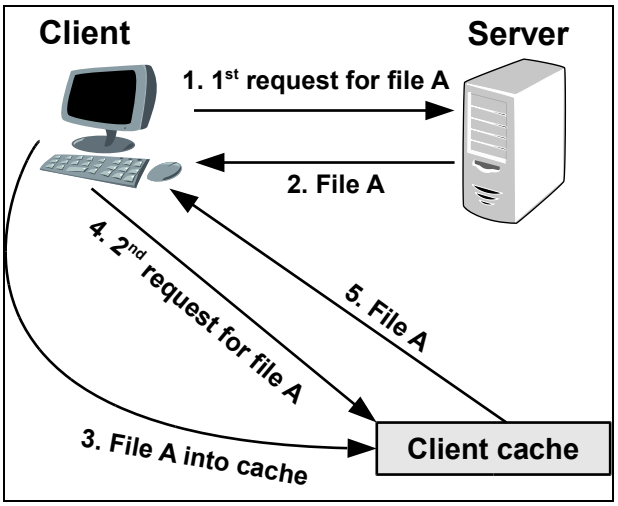

Figure 2.5: Client-side caching

If there are caches on both sides of communication, the server cache-miss ratio increases. Clients often request files in their own cache, so they do not need the server to get the file. On the other hand, the server gets requests on different files, so the server-side cache is useless. This case is described in [**22**].

2.4 Security in Traditional DFS

Security in traditional DFS is solved by using an authentication, an authorization, and a controlled access to files. All mentioned traditional DFS can use a Kerberos system as an authentication protocol.

The Kerberos is a network authentication protocol which allows the users to authenticate themselves to someone else. The Kerberos prevents nodes in DFS from an eavesdropping or a repeating authentication communication and guarantees a data integrity. It was primarily made for client-server model and provides a mutual authentication: both client and server verify each other identity [**23**].

The Kerberos is based on the Needham-Schröder's protocol and Key Distribution Centre (KDC). The Kerberos consists of two logical independent parts: an Authentication Server (AS) and a Ticket Granting Server (TGS). The Kerberos uses a ticket which serves for providing users' identity. KDC holds database of secret keys and every user also knows this secret key. This key is used to authenticate user's identity. For communication between two subjects KDC generates a session key for this communication [**23**].

Kerberos uses Single Sign-on Mechanism which allows logged in users using several services without the need of logging in to all these services again. After the user has logged in, the user becomes a Ticket Granting Ticket (TGT). When users want to access services, they use TGT to get a Service Ticket (ST). ST is always sent while user requests a service. ST has a valid time. The ticket must be refreshed after expiration [6]. Whole authentication procedure is depicted in Fig. 2.6.

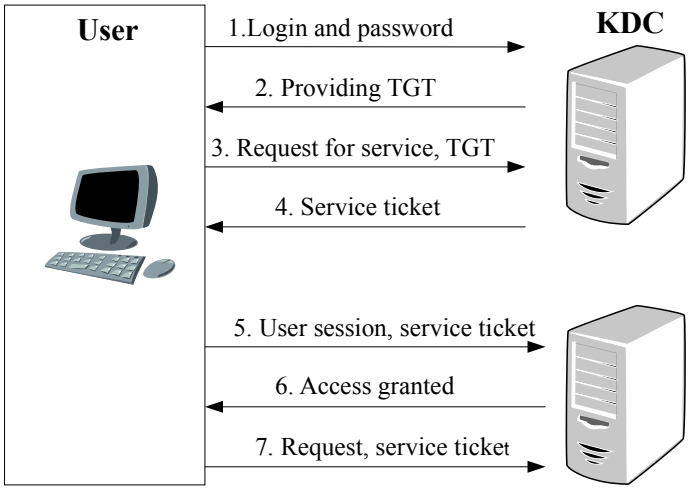

Figure 2.6: Kerberos authentication procedure

All traditional DFS use Access Control List (ACL) to control access to files. ACL defines a list of rights for all files and directories. This list assigns users and rights for the file or for the directory. Users can access files after they authenticate themselves and have adequate rights to these files. There can be some problems when using ACL (e.g. user must have same ID in the whole system). This problem solves for example OpenAFS by using extended rights for files, and separated user's management [**12**].

2.5 New Trends in Distributed File Systems

In this subchapter, we provide information about new trends in distributed file systems. We explore algorithms that provide reliability, increase performance and security in DFS. We start with algorithms which are used for achieving reliability.

2.5.1 Reliability

For achieving reliability in distributed file systems, the first step is choosing a suitable file system for storing data and/or metadata. As was mentioned in section 2.2.2, reliable file systems usually use journaling. The journal technique can be also used for keeping whole DFS consistency.

The second step in increasing reliability is using a file replication. A file replication can be also used in achieving better system performance. We discuss the difference between replication for performance and replication for reliability in section 2.5.2.

Journaling

A journal in DFS can be used as a write-ahead commit log for changes to the file system that must remain unchanged [**24**]. Every transaction made by a client is recorded in the journal, and the journal file is flushed and synchronized before the change is committed back to the client [**24**].

The journal can be also used as a checkpoint [**24**]. The checkpoint is a summarization of all records that were written to the journal, and presents a consistent state of the DFS. When any fault appears, the DFS can recover by returning to this checkpoint by applying changes which are stored in the journal. Journal and checkpoint serve for maintaining metadata information. If journal records or/and checkpoint are missing or are corrupted, namespace information is lost (in this concept). To prevent these incidents, critical information about namespace is stored on several storages.

Replication

Recall that replication in a traditional DFS is usually made administratively. A nowadays technique of replication is a dynamic replication presented in [**25**] or [**26**]. Both papers use statistical information about files and servers. Based on these statistics, it can be decided which data will be replicated and where. A replication algorithm is depicted in Figure 2.7. Most of novel approaches use the file as a basic replication unit.

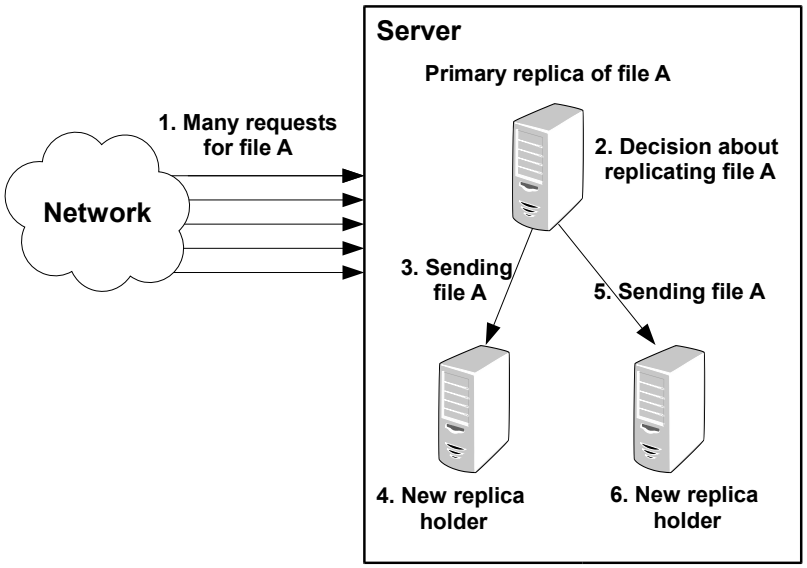

Figure 2.7: File replication

In the case which is depicted in Figure 2.7, many remote users request the file A. If the number of requests reaches the threshold value, the replication process starts at the Node 1 (Primary replica holder). The file A is then sent to the new replica holder (Node 2). This node becomes new replica holder. Simultaneously, the file A is sent to Node 3, which becomes new replica holder too. After replication process ends, all three nodes can provide file A.

The Criteria Based Replication (CBR) is a dynamic file replication model [**25**]. It uses two main algorithms for achieving high availability and reliability. The first algorithm is the Criteria Based Replica Placement algorithm (CBRP). This algorithm collects information about physical location of the server, load of the server, etc. The second algorithm is the Criteria Based Primary-copy Assignment (CBPA). This algorithm is used for choosing the primary copy (or replica). CBPS is used for making primary copy available for maximum number of clients at any single session. Statistics for these two algorithms are collected through system calls and by predefined system variables.

The Criteria Based Replica Placement algorithm monitors each criterion individually. Then, it periodically calculates the result for each criterion. If this result exceeds a threshold value, the file is replicated.

In the Criteria Based Primary-copy Assignment algorithm, the server is being chosen for holding the primary replica of a file. This algorithm makes a list of

servers with chronological priority to be a primary-copy server [**25**]. The decision of choosing primary copy server shall be done before client requests for a file. This file is then highly available for the client.

Another way in dynamic file replication is storing the information about a whole system such as the service ratio of peers, reliable value of peers, etc. [**26**] Based on this information, the system can place a replica at the most reliable place at a particular time. The whole system in this conception is divided into *peers* and *super peers*. Super peer is a computer which is rich in resource and capability, and is used to manage peers in its group.

Super peer also collects statistical information about peers in group and runs replica management service. This service has fully knowledge about master replica location, network topology and bandwidths to the relevant peers [**26**]. The decision for making the new replica and the new replica placement is made by the super peer. The super peers maintain a list of frequently requested files, and also collect information about average response time. This list is periodically updated. If the response time of any file exceeds threshold value, the file is replicated.

The previous two papers [**25**] and [**26**] present methods for file replication. These two papers do not mention algorithms which can be used for updating files and their replicas. The file replication can be used for achieving reliability too [**27**]. In addition to [**25**] and [**26**], it presents the master-slave full replication method. This method updates the master replica first; other replicas are updated only when there is a need to receive the data from them. For achieving data integrity, this DFS uses content hash. Master-slave replication is used for increasing performance and decreasing response time.

The replication of the data can serve not only for achieving reliability and availability, but also for providing Quality of Services (QoS) [**28**]. Not all replica holders can provide the data as fast as the others. Also, the speed of the network can change because of sharing the link among users. The QoS algorithm proposed in [**28**] selects the most appropriate resource from three different perspectives. The first perspective is the available bandwidth; the second predicts the possible trend

of bandwidth utilization. The last factor is the bandwidth requirement for the currently requested file.

Another way of storing file content is based on a P2P network [**29**]. An example of file storage is depicted in Fig. 2.8.

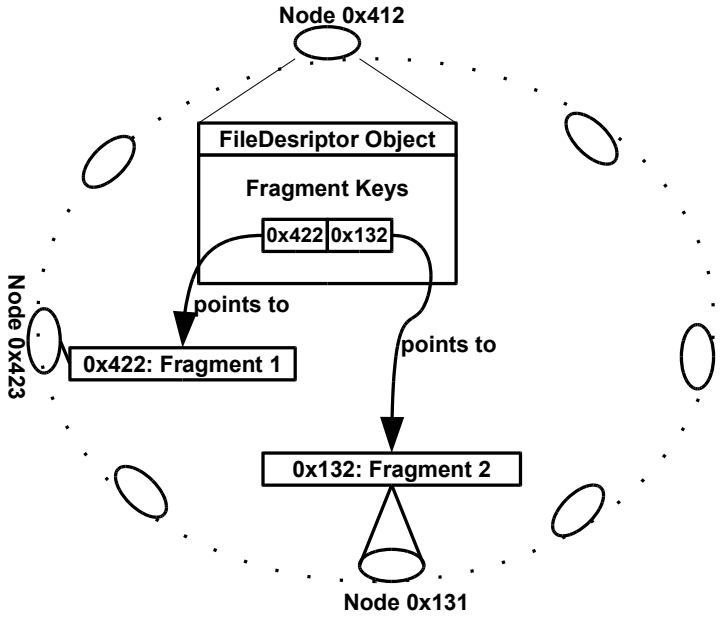

Figure 2.8: Example of file storage in the P2P overlay [**29**]

Files in this system are split into fragments, which are then stored on clients (peers). Links to these fragments are stored in a Distributed Hash Table (DHT). The DTH also maintains file and directory descriptors. Every client in this system is responsible for files, whose fragment keys are near the peer ID. A fragment key is made from a 160-bit random value and a combination of a path name and a fragment number as a domain [**29**]. File descriptor is responsible for storing the fragment keys.

Advantage of using path name calculating hash is in updating file content. If new data is written to the fragments, there is no need to update DHT records. Fragments are still on the same nodes [**29**]. DHT records must be updated only when the file is moved between directories. In this circumstance, file fragments must be moved to new nodes according to the new calculated fragment keys.

This DFS uses two ways in file replication: passive and active. Passive file replication is done by using file caching. In the cache, frequently asked fragments are temporarily stored. Active file replication works as follows. The peer, which ID

is closest to the fragment ID, is responsible for a fragment replication [**29**]. This peer checks periodically if there are enough replicas in the system. If any of the peer changes its status to *leave*, the others peers must check if they are now responsible for the fragment.

File replication for achieving reliability is also used in other DFS. CloudStore [**30**] typically uses 3-way file replication, but administrator can set up to 64 replicas of one file. If there is a need for replication (e.g. node outage), a metadata server can replicate a file chunk to another node. This conception of file replication for achieving reliability is derived from the Google File System [**31**].

Reliability in GlusterFS [**32**] uses three file distribution methods. The first one is Distribute-only. In this concept, each file is stored only once. This solution does not provide increased reliability. Reliability can be achieved by using the second method: distribution over mirrors. This means that each storage server is replicated to another storage server. The third method is stripping large files over nodes in DFS. This is usually used for very large files (minimum is 50GB per file). The first and the third method are less reliable then the second method.

For achieving high availability and reliability, one server (primary server) can be replicated to another server (hot standby server). When the primary node fails, the hot standby server can be switched to an active role [**33**]. Authors of [**33**] also describe how to maintain block locations and lease for hot standby server (they replicate messages from DataNodes to the hot standby server) and how the hot standby server assumes the active role (they use Apache ZooKeeper [**34**]).

File Locking

While achieving reliability in DFS, the data consistency is also very important. The data consistency can be achieved by using file locking. There are usually two types of locks: a shared lock for file reading and an exclusive lock for file writing. A shared lock is used when the data can be read simultaneously by many clients, but cannot be written simultaneously. While a file is being read by clients, other clients cannot write into this file. An exclusive lock means that only one client can access a locked file and can write file content.

The DFS can be based on mobile agents [**35**]. The mobile agents are used for communication, synchronization, data access, and for maintaining consistency by using two layer lock request mechanism. In this system, there exist four kinds of agents:

- Interface Agent (IA) accepts and process file systems calls from client, and coordinates with others agents.
- Working Agent (WA) accept calls from IA, and executes file operation on remove server (WA moves itself to this server).
- Domain Manage Agent (DMA) is responsible for a cache, name, a space management and an access control management in the domain [**35**].
- Main Management Agent (MMA) is responsible for a coordination of DMAs.

There is one Main Consistence Server (MCS) with one MMA and many Domain Consistence Servers (DCS) with theirs DMAs in this conception. For the file locking, there is a two-layer lock request mechanism. The two-layer lock system is depicted in Fig. 2.9. Obtaining a file lock is done by asking DCS for it (IA→WA→DMA), if DMA does not have the requested lock, then the lock must be obtained at MMA (DMA→MMA).

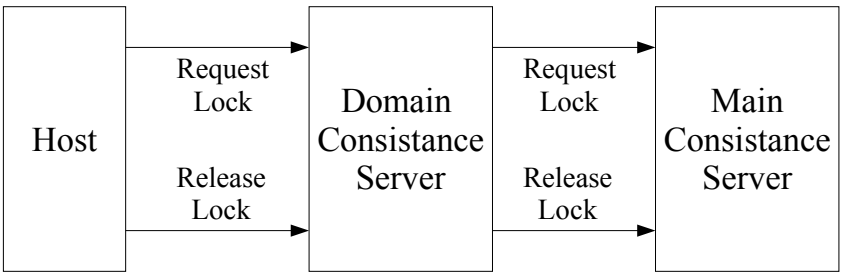

Figure 2.9: The two layer lock request mechanism [**35**]

2.5.2 Increasing Performance

This section describes modern trends in DFS with a focus on increasing performance. First of all, we describe trends in the data storage and the metadata storage organisation. Data storage is used for storing file content. Metadata storage usually stores file attributes and links to the file content in the data storage. Afterwards, we focus on algorithms which are commonly used for increasing performance: replication and caching algorithms.

Data Storage

Data storage is used for storing file content. When users want to upload a file to the DFS, they send the entire file to the server. On the server side, this file is split into two parts: file content and file metadata. File content is then stored in a data storage node. Uploading a file to a server and storing file content is depicted in Fig. 2.10.

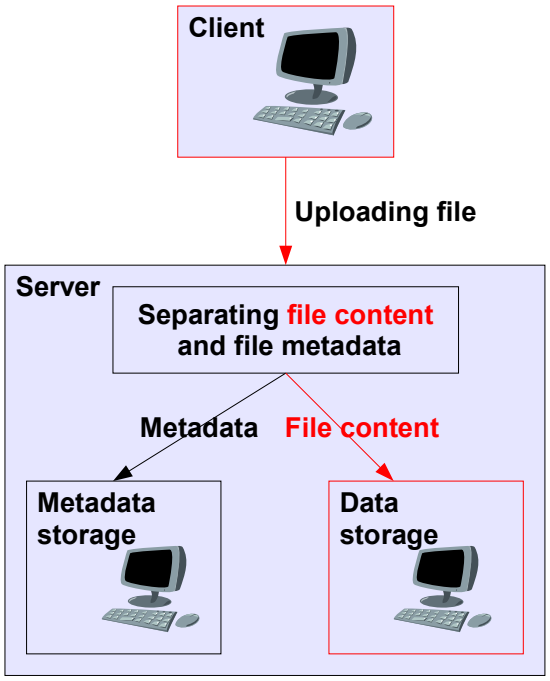

Figure 2.10: File uploading process – file content

For the data storage, local hard disks with their own file systems are usually used. On nodes with OS Microsoft Windows, NTFS is commonly used. In UNIX-like systems, several different file systems exist. Not all of these systems are suitable for all types of files.

According to the tests in [**36**], the ReiserFS is more efficient in storing and accessing small files, but it has a long mount time and is less reliable than EXT2/3. XFS and JFS have good throughput, but they are not efficient in file creation. EXT2/3 has severe file fragmentation, degrading performance significantly in an aged file system [**36**]. The decision on which file system will be used for data storage is an important part of DFS design.

Another method of storing file content is a designing new data organization of a hard disk. This concept is used when existing data organization (file system) of

a hard disk is not suitable for files which will be stored there. A new hard disk organization [**37**] is depicted in Fig. 2.11.

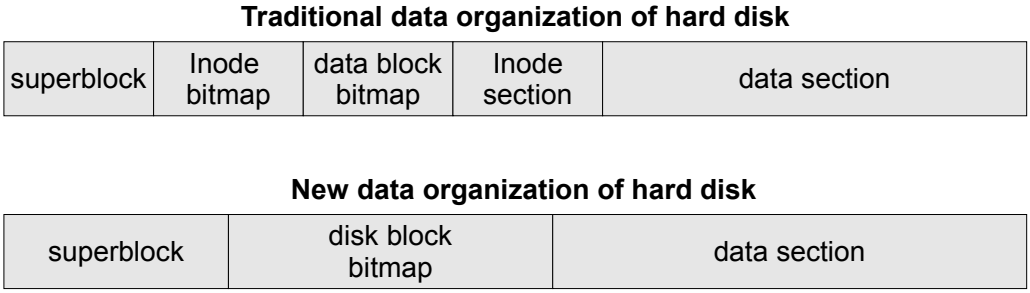

Figure 2.11: New data organization of hard disk [**37**]

In this proposal, the superblock is located at the beginning of the file system. Next to the superblock, the disk block bitmap is situated. There is no need for the *inode* section, because inodes are spread over the entire disk. The disk block bitmap serves for recording weather a block is used or not [**37**]. Every single inode can be locked without increasing the total amount of locks. Distribution of the inodes also makes the system more expandable because the distribution makes it possible for the number of inodes to increase or decrease dynamically [**37**].

Writing and reading file content or creating a new file is a slow operation. I/O operations are the bottleneck in achieving better performance in DFS. Uploading and storing files on a server have several steps. These steps must be done chronologically to ensure data consistency. The steps are: sending a file to the server → splitting file content and file metadata → creating a new metadata record → creating a new file and storing file content → setting file attributes → connecting metadata with the file handle. Both, creating the metadata record and storing content, are slow operations. These slow operations can be accelerated [**38**] [**39**].

The performance can be increased by making changes in an upload protocol [**38**]. These changes can be made in different ways:

- *Compound operations*. In this proposal, authors suppose that the steps in upload protocol are independent from the others. So, the steps can be done in parallel. E.g. a new metadata record can be created and the attributes can be set in one step. This reduces the amount of sent messages during upload process.

- *Pre-creation of data files* at the data storage servers. Creating a new file and getting a file handle for connecting with a metadata record is a slow operation. The file handles can be created before the file is uploaded to the server and then the file can be uploaded and the metadata record can be made parallel.
- *Leased handles*. In this proposal, a client has leased I/O handles (from a data server). If the client wants to upload a file to the server, the client application can use one of the leased I/O handles. Creation metadata record and file uploading can be done in parallel.

The performance can be also increased by using methods which presume uploading huge amount of small files [**39**]. Method pre-creating file object is similar to [**38**]. Other methods are:

- *Stuffing*. While using this method, the first block of a new created file is stuffed with stuffing bits. This concept supposes that the uploaded files are small. The client application can create a metadata record in parallel to uploading file content. There is no need to allocate more blocks on the hard disk.
- *Coalescing Metadata Commits*. If the client application uploads many small files, creating and storing metadata records takes long time. At the metadata storage, new metadata records can be collected and flushed into database periodically or after reaching threshold value. This proposal decreases time which is necessary for creating metadata records.
- *Eager I/O.* While uploading a file to the server, the system usually sends two messages. The first message is for creating a file handler (answer to this message is file handler); the second message is a file content. If we presume that the client application always gets file handler, these two messages can be merged into one. This proposal saves one message.

Both [**38**] and [**39**] demand cooperation between the file storage nodes and the metadata storage nodes.

Papers [**36**], [**37**], [**38**] and [**39**] assume that the whole file is stored in one node. Another way of storing files is splitting a file content into file fragments and

storing these fragments on the client side. This proposal does not work on a client-server model, but works in P2P networks. Links to the file fragments are stored in a distributed hash table. The entire system is described in section 2.5.1. The P2P system architecture is depicted in Figure 2.8.

Dynamically upgrading, replacing, and adding new storages in the system and file operations like creating, deleting, and appending can result in load imbalance. In this case, the file chunks are not distributed as uniformly as possible in the nodes [**40**]. To avoid load imbalance, the file chunks can migrate in the system especially from the heaviest nodes to the lightest nodes [**40**]. For reallocating of the file chunks, two properties of the nodes have to be fulfilled: low movement cost and fast convergence rate.

Metadata Storage

Metadata are a specific type of data which hold information about a certain item's content. In DFS, metadata are used for providing information about files which are stored in the data storage. This information is usually the date and time of file creation, the date and time of the last modification, the file size, the file owner, the file access rights, etc. The file attributes are a part of these metadata.

Metadata storage also provides information about a directory structure. All this information is created during the upload process (see Figure 2.12). Each record must also have a link to the data storage. Metadata storage must provide functions for getting and storing file metadata, file searching, moving files within directories, deleting files and creating files. Additionally, metadata storage can provide locks for ensuring consistency during the file access. Metadata is usually stored in a database or in a tree.

Database records are used, e.g., in the AFS. While using the database, all metadata operations are represented by a database query. Trees are used e.g. in [**41**] or [**24**]. Entire tree is usually stored in RAM. Tree is used for maintaining namespace information. Adding a new file metadata record is simply adding a new node to the tree. This node must also have a link to the file content in the data storage. If the system uses file replication, then the node corresponding to the file

must have links to all replicas. Furthermore, the metadata server can provide load balancing while choosing suitable data storage for the clients.

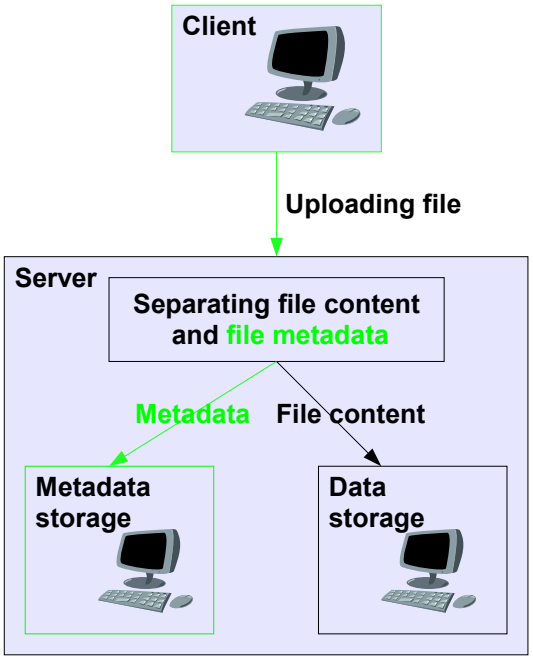

Figure 2.12: File uploading process - metadata

In [**41**], DFS uses one metadata server per cluster. This metadata server manages all file system metadata. Additionally, metadata server provides garbage collection for orphaned files, and provides services for file migration. Metadata server in this solution communicates with the data storage, gives the data storage information and collects statistical information about these servers. All metadata are stored in RAM (for increasing performance), and on hard disk (for increasing reliability). When the client requests a file, metadata server returns links to all data storages where file replicas are stored. Client stores this metadata in cache.

If the client wants to read a file, the metadata server returns a link to the data storage which is the closest to the client [**24**]. The metadata node keeps a list of available data nodes by receiving heartbeat messages from these nodes. When a client wants to upload a file to the DFS, metadata server nominates three data storage for storing file content. The client must then arrange the file replication.

For accelerating metadata operation, some metadata can be stored at the client side [**42**]. By using cached metadata, the communication with metadata server is not necessary. The typical metadata operations can be run locally. The

disadvantage of this approach is in consistency of cached metadata. To overcome this disadvantage, delegation scheme is employed to guarantee metadata cache consistency [**42**].

Another way of storing metadata involves using a log-structured merge tree (LSM). LSM tree is multi-version data structures composed of several in-memory trees and an on-disk index [**43**]. The database consists of a set of indices, a log manager, and a checkpointer. Indices are data structures which are optimized for searching and storing database records. The log manager is used for persistently logging database modifications. Database log can be used for restoring database when the system crashes. The example of the database is depicted in Fig. 2.13.

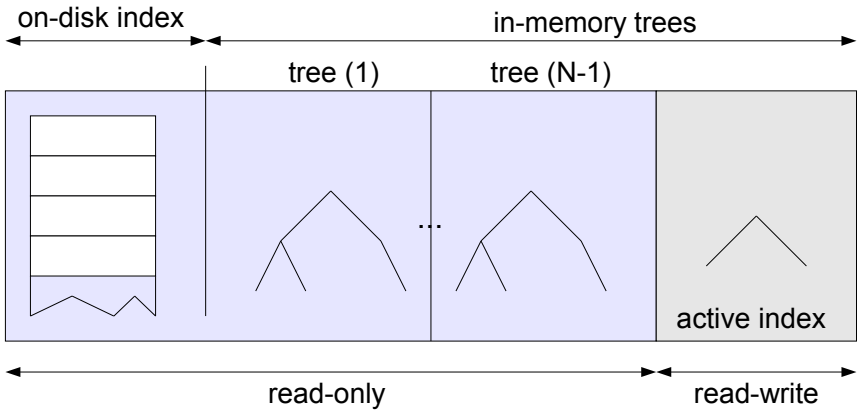

Figure 2.13: Database with N in memory trees and on-disk index [**43**]

In the database, an index consists of a list of *N* in-memory trees and a single on-disk index. The changes to the metadata are inserted to the *active* tree (last tree). All of the other trees and on-disk index are read-only. While looking for a record, the system searches through all in-memory trees from *N* to *1*. If the record was not found, the system would look into on-disk index. This method provides latest version of a record.

Caching

Recall the section 2.3.2, a cache in the computer system is a component which stores data that can be potentially used in the future. The cache has limited capacity hence exist caching policies which try to predict future requests.

Caching policies mark the entity which can be removed from the cache when a new entity comes to the cache. Most of these algorithms are based on statistics made from previous data requests. These policies can be divided into simple policies, statistical based and hybrid policies. Simple policies do not use any statistics, statistical based policies use statistics from previous accesses, and hybrid policies use a combination of other policies to make a replacement decision. Several researches are oriented on developing new caching policies. State-of-the-art in caching policies is provided in section 4.

All caching policies attempt to act like *OPT*. The OPT (Optimal) strategy chooses a data to be removed from the cache based on when it will be used again in the future [**44**]. The data that will be used farthest in the future will be removed first. Thus, this policy will have the best cache read hit ratio which is a ratio of cache hit count to total number of all requests to data. It is also the most effective replacement policy, but it cannot be implemented in practice since that would require the ability to look into the future.

A decentralized collective caching architecture is presented in [**45**]. In this concept, caches on the client side are shared among clients. The whole file is used as a basic caching unit. When a client downloads a file, this file is then stored in the client's cache. The server stores a list of clients for each file. This list also contains the client network address. When another client wants to access this file, the server returns this list. The client can then download the file from one of the listed clients. The server then adds this new client to the list. This proposal decreases server work-load, but it requires cooperation among clients. Collective caching can be also used in mobile ad hoc networks [**46**].

Collective caching architecture provides close-to-open consistency. Central server maintains commit timestamp (logical clock per every shared file). This number is increased every time a client commits new file content. When a client wants to download a file, a client application gets timestamp and a list with other clients holding requested file in their caches. Then, the client application looks into its own cache whether it has the file. If the file is found in the cache, the timestamp

is verified. If the file is old, new content is downloaded either from other client (if any client has the file) or from the server.

Another way to reduce server workload and network bandwidth is by using proxy caching. Proxy caching in the DFS was introduced in [**47**]. A proxy cache stores data which are requested by clients. The architecture of proxy caching is depicted on Fig. 2.14.

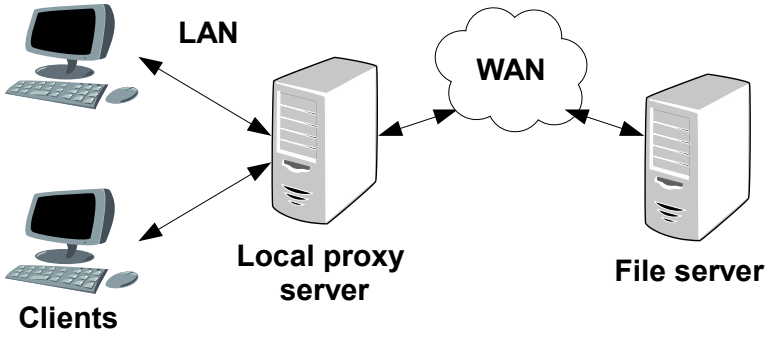

Figure 2.14: Local proxy caching

The whole cache is stored in a proxy server. The proxy server in this paper is on a local network. This paper also assumes that the connection to the server is slow (typically uses WAN). All file requests to the remote server pass through this proxy server. If the requested data are in the proxy cache, the proxy server returns requested data and there is no need to communicate with the remote server. Compared to the collective caching, proxy cache is a cache which is shared among users on local network segment.

2.5.3 Security

Modern trends in achieving security in DFS can be divided into two parts: secure network communication and secure file storing. In this section, modern trends in these areas are explored.

Secure Communication

Secure network communication protects data against eavesdropping attacks. The communication is usually encrypted by using a symmetric cipher. This cipher needs a key, which is used for data encrypting. The same key is also used for data decrypting.

In [**48**], a session key is used for ciphering. The session key is a unique key, which is established each time the session is created between server and client. This paper also presents techniques which prevents other kinds of threats like user masquerading, man-in-the-middle attack and replay attack. User masquerading is prevented by using unique credentials, which are selected randomly by the file server and established during a genuine user login session [**48**]. Man-in-the-middle attack is prevented by checking the integrity of messages by applying message integrity codes [**48**]. Replay attack is prevented by using a secure communication channel.

Another way of creating keys is presented in [**49**] where Identity Based Encryption (IBE) is used. This algorithm allows deriving a public key from some known aspects of user identity. This information can be, for example, IP address, email address, validation period, etc. IBE also needs a third party entity called a Private Key Generator. This entity provides a private key after authentication for decrypting data. However, this entity is also a bottleneck in key distribution. This restriction is eliminated by allowing each client and server to maintain own IBE infrastructure [**49**].

In [**50**], a DFS which is based on Java and Java Remote Method Invocation (RMI) is presented. RMI is used as a middleware mechanism for remote file communication [**50**]. For secure communication, Java Crypto libraries are used. These libraries perform block level encryption of file content and also serve for communication with the server.

Secure File Storing

File content which is stored on data nodes can be stored on a traditional file system or on a secured file system. Storing data on a secured file system allows one to protect data against theft. There are several Encrypting File Systems, such as TransCrypt, NCrypt, eCryptfs, dm-crypt, Microsoft EFS, etc. Some of them are described in [**48**] and [**51**]. In [**48**], TransCrypt is used for data encrypting and decrypting. TransCrypt is an enterprise-level, kernel- space encrypting file system for Linux.

In [**51**], secure file storing which uses a symmetric cryptosystem and a per-file encryption key is presented. These encryption keys are stored in Meta-Data Storage (MDS). According to this paper, MDS is trustworthy for maintaining cryptographic keys. As the encryption algorithm, an Advanced Encryption Standard (AES) block cipher in Cipher Block Chaining (CBC) mode is used. Unauthorized modification of a file is prevented by using a cryptographic hash or "digest" of every file. As a hashing algorithm, Secure Hash Algorithm (SHA) family of hash function is used and produced hash is stored in MDS.

Paper [**52**] presents a DFS which is based on a P2P network. The file saving process is depicted in Figure 2.15.

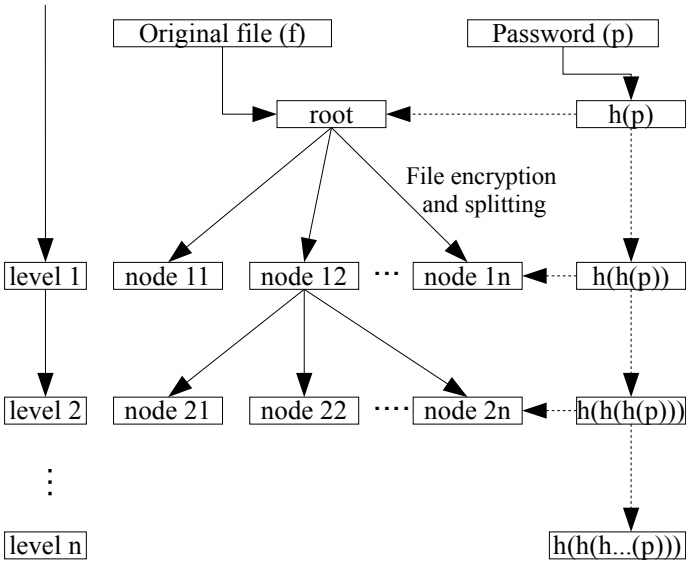

Figure 2.15: Hierarchical file encrypting [**52**]

Peers in this network are used for storing encrypted files. As a cryptographic key, hash is used. This hash is derived from the user password. As a hash algorithm, SHA-256 is used. Encryption is done on N levels, where on each level the file from the previous level is split into pieces. These pieces are encrypted by using a new key and are sent to the next level. A new key is created by hashing hash from the previous level. On every level, nodes store hash which is used for encryption and links to nodes where file pieces are sent. This is done for future decryption. When the final level is reached, file pieces are stored on nodes. All file pieces are N-times encrypted before storing.

2.6 Motivation for Research

DFSs provide many advantages for users that use them as remote data storage. These advantages are reliability, scalability, capacity, security, etc. However, none of described concepts of DFSs considers mobile clients. Accessing files from mobile devices has to take into account changing communication channels caused by the user's movement.

Traditional DFSs that are widely used were designed before mobile devices spread. Now, it is hard to develop mobile client applications and to implement algorithms for mobile devices into these DFSs. Even, none of the current DFSs, InterMezzo, BlueFS, CloudStore, GlusterFS, XtreemFS, dCache, MooseFS, Ceph and Google File System, has suitable clients for mobile devices [**53**] [**54**] [**55**]. Based on these observations, we decided to aim our research at mobile devices. Specifically, our research is oriented on increasing efficiency of remote accessed data and on maintaining data consistency in these devices.

3 KIV-DFS

In this section, we briefly describe KIV-DFS. KIV-DFS is an experimental distributed file system which is being developed at the Department of Computer Science and Engineering, University of West Bohemia. KIV is an acronym for Czech name of the department (Katedra Informatiky a Výpočetní techniky). This DFS is designed to support mobile devices. In this section, we introduce the system architecture and we also describe modules which the system consists of. The whole system is described in [**56**] in more detail.

3.1 KIV-DFS Architecture

The KIV-DFS distributed file system consists of two main parts: server and client application. The system architecture is depicted in Fig. 3.1. The modules at the server side communicate via KIVFS protocol which is based on TCP protocol. This allows running individual modules on different machines where each module runs as an independent process. Of course, several modules can run on one node which increases availability. Next, the individual modules are briefly described.

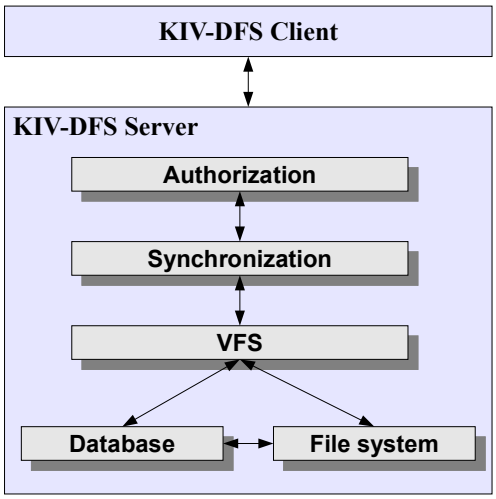

Figure 3.1: KIV-DFS architecture

3.2 KIV-DFS Client

The client module allows client to communicate with KIV-DFS servers, and to transfer data. The client application exists in three main versions: standalone application, core module of operating system and Filesystem in Userspace (FUSE).

The client communicates with the Authorization Module. For communication between client and server, any network providing TCP/IP stack can be used. For ensuring security of transferred data, TLS (Transport Layer Security) encryption is employed.

Before the users can access the data, they have to authenticate themselves. The Kerberos system serves as an authentication authority. After the authentication is approved, the server checks the user name. If the verification of the user name succeeds, the user can access the server.

3.3 Authorization Module

This module is an entry point to the system. It ensures authorization and secure communication with clients. The communication channel is encrypted by using OpenSSL. The server maintains the database of the users who can access the system. This database can differ from Kerberos database. The request for authentication to the Kerberos system and the authorization to the user database is done simultaneously. After successful connection, the proxy module establishes the connection with the synchronization module.

3.4 Synchronization Module

The synchronization module is a crucial part of the whole system. Several clients can access the system via several nodes. Generally, different delays occur in delivering the messages. The KIV-DFS system uses Lamport's logical clocks [**1**] for synchronization. Every received message gets a unique ID corresponding to the logical clock. The synchronisation is based on this unique ID, which serves as a timestamp. This ID is also used for synchronization among nodes.

After the negotiation of the unique ID among nodes, the message is sent to all nodes participating in the system. For storing messages (requests), each node contains the queue which is stored in the database (database module). The requests from the queue are forwarded to the VFS layer ordered by logical clock values.

Only after all nodes processed the request to VFS, the file can be provided. By updating all replicas before providing the file, KIV-DFS meets the requirements

of strong consistency. The configuration of other servers and the database are checked at the start time of the system.

3.5 Virtual File System (VFS Module)

The VFS module hides the technology used for data and metadata storing. Based on the request, the module determines whether it is aimed at the metadata, e.g. to list the directory, create a new directory, or is aimed at the file access. Then, the request is sent to the DB module or to the File System module.

3.6 Database Module

The Database module stores metadata, the list of authorized users, and the client request queue.

Metadata contain all information about files, such as names, the location in the directory structure, ACL information, size, and the physical location of the file.

For increasing metadata performance such as directories browsing and creating, directory or file movement and so on, the database is stored in main memory. The database is replicated on each node for increasing availability and stability.

The synchronization of databases is solved at the synchronization level of KIV-DFS. It ensures the independence of the replication and synchronization mechanisms of databases stored on different nodes. Thus, the database can be easily migrated among nodes. The database is designed in a minimalistic way. It does not contain unnecessary data that are not required. The ERA diagram and individual tables are described in [**56**] in more detail.

We focus here on the table *replicas* which contains attributes that are used in sections 5 and 6. This table contains the information about the state of the files stored in individual volumes. The lock is set for all replicas when the file is created or modified. The lock is removed for the individual replica only, when the file is successfully uploaded and the whole synchronization is over. The next three parameters, not common in file systems, are *file.version*, *file.readHits* and *file.writeHits*. The value file.version provides the version of the file, which is an

integer counter, increased by one by each update. The value file.readHits is also an integer value and is increased when the file is accessed. The value file.writeHits is an integer value and is increased when the file is updated. Both these counters are periodically divided by 2. This ensures that files read more recently, though less frequently read, will have higher counter value over files more frequently read in the past. The default aging period is set to one week.

3.7 File System (FS Module)

The File system module serves for storing file content on physical device like hard disks. It is utilized to work with the content of the files that the user works with.

The FS module also manages active data replication. The FS module starts the replication of the file in the background. When the file operations are performed, the replicas are locked at the metadata level (Synchronization layer). This prevents a situation of simultaneous file access. The metadata record is unlocked after the file is stored. Similarly, when the replication is finished, the metadata record of replicas is unlocked. By using this approach, the KIV-DFS supports multi-RW replication.

4 Caching Policies

As was written above, the KIV-DFS is designed to support mobile devices. At the server side, the support means that all overhead are located at the KIV-DFS server and all metadata are kept on servers. Thus, mobile devices can fully use their capabilities to run user's application which accesses remote files.

Use of mobile network for accessing remote files is bottleneck of the system. The connection to the remote system can vary (see 1.3). The highest speed of the mobile network is often limited by the use of the Fair User Policy (FUP) by the mobile connection provider. The FUP restricts the quantum of the downloaded data in a period of time [**57**]. So far, users access the same data repeatedly; we can use a cache to increase system performance. The cache on the mobile device can be stored on memory cards. The use of the cache in a mobile device is depicted in Fig. 4.1.

In this chapter, we firstly provide state of the art in caching policies. Then, novel caching policies are described.

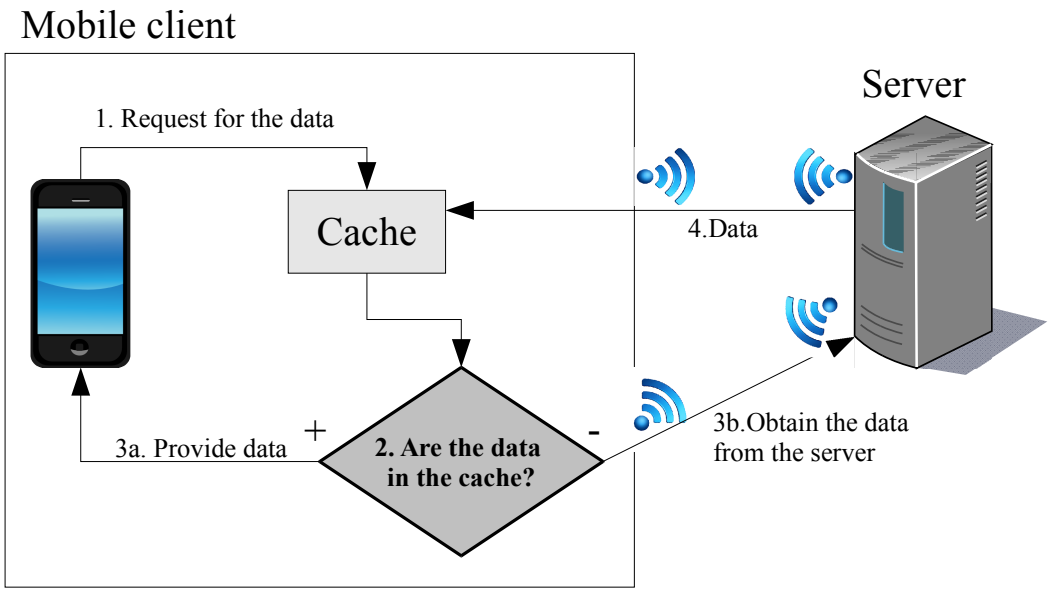

Figure 4.1: Cache in a mobile device

4.1 State of the Art in Caching Policies

The cache stores content that was requested in past in order to provide this content faster in the future when the content is requested again.

Cache uses caching policy to make a replacement decision when the cache is full. We describe replacement policies which are commonly used in distributed file systems or in operating systems.

Clearly, an optimal replacement policy (OPT is described in section 2.5.2) replaces data whose next use will occur farthest in the future. However, this policy is not implementable. We cannot look into the future to get needed information about usage of the data. Hence, no implementable caching policy can be better than an optimal policy. OPT policy can be used ex-post only for comparison to other policies.

Caching policies can be divided into three categories: simple, statistics-based and hybrid.

4.1.1 Simple Caching Algorithms

Simple caching algorithms do not use any statistics or additional information. For replacement decisions, they usually employ other mechanisms. Examples of simple caching algorithms are Rand, FIFO, FIFO with 2nd chance, and Clock. None of these caching policies takes user behaviour into account.

FIFO

First-In First-Out is the first simple replacement policy. The data that are chosen to be replaced are the oldest in the cache. Data in the cache are ordered in a queue. The new data are placed on the tail of the queue. When the cache is full and new data come into the cache, the data from the head of the queue are replaced [**58**].

FIFO with 2nd Chance (FIFO2)

First-In First-Out with second chance is a modification of the FIFO caching policy. FIFO2 stores the data units in a queue. In contrast to FIFO, FIFO2 stores a reference bit for each data unit in the queue. If a cache hit occurs, the reference bit is set to 1. When a replacement is needed, the oldest unit in the cache with a reference bit set to 0 is replaced and the reference bit of the older units is set to 0 at the same time [**59**].

RAND

RAND or Random is a simple replacement policy which chooses data to be replaced based on random selection [**44**]. It is very easy to implement this replacement policy.

CLOCK

The CLOCK replacement policy stores the data units in a circular buffer [**60**]. Clock stores a reference bit for each cached unit, and a pointer into the buffer structure. If a cache hit occurs, the reference bit of the requested data unit is set to 1. The data to be replaced are chosen by circularly browsing the buffer, and searching for the unit with a reference bit set to 0. The reference bit of each data unit with the reference bit set to 1 found during the browsing is reset to 0 [**61**] .

4.1.2 Statistics-based Caching Algorithms

Statistics-based algorithms employ statistical information about data in the cache: frequency of the accesses, and recency of the last use of data. Frequency is used by an LFU algorithm and recency by LRU and MRU algorithms.

LRU

The Least Recently Used replacement policy uses the temporal locality of the data [**44**]. Temporal locality means that the data units that have not been accessed for the longest time will not be used in the near future and can be replaced when the cache is full [**62**]. According to the tests [**63**], LRU seems to be the best solution for caching large files. LRU is frequently implemented with a priority queue. Priority is the timestamp of last access. The disadvantage of the LRU policy is that the data unit can be replaced even if the unit was accessed periodically many times. In this case, the file will probably be requested in the near future again.

LFU

The Least Frequently Used replacement policy replaces the data that have been used least. For each data unit, there is a counter which is increased every time the

data unit is accessed [**44**]. The disadvantage of this approach is that the data units in the cache that have been accessed many times in a short period of time remain in the cache, and cannot be replaced even if they will not be used in the future at all.

MRU

The Most Recently Used replacement policy works conversely to LRU. MRU replaces the most recently accessed data units. MRU is suitable for a file which is being repeatedly scanned in a looping sequential reference pattern [**64**].

4.1.3 Hybrid Caching Algorithms

The combination of LRU and/or LFU with other replacement policies results in hybrid algorithms. These algorithms mostly combine LFU and LRU to get better results in the cache hit ratio.

2Q

2Q replacement policy uses two queues. The first queue uses the FIFO replacement policy for data units and is used for data units that have been referenced only once. The second queue uses LRU as a replacement policy, and serves for so-called hot data units. Hot data units are units that have been accessed more than once. If a new data unit comes to the cache, it is stored in the FIFO-queue. When the same data unit is accessed for the second time, it is moved to the LRU-queue. Author recommends setting capacity of FIFO to 50% of cache capacity [**65**].

LIRS

LIRS replacement policy uses two sets of referenced units: the High Inter-reference Recency (HIR) unit set and the Low Inter-reference Recency (LIR) unit set. LIRS calculates the distance between the last two accesses to a data unit and also stores a timestamp of the last access to the data unit. Based on this statistical information, the data are divided into either LIR or HIR blocks. When the cache is full, the least recently used data unit from the LIR set is replaced. LIRS is suitable for use in virtual memory management. Author recommends setting capacity for a LIR section to 90% [**66**].

MQ

MQ replacement policy uses multiple LRU-queues. Every queue has its own priority. The data units with a lower hit count are stored in a lower priority queue. If the number of the hit count reaches the threshold value, the data unit is moved to the tail of a queue with a higher priority. When a replacement is needed, the data units from the queue with the lowest priority are replaced. Author recommends setting the number of queues to 5, and capacity of out queue to 10 [**67**].

FBR

FBR replacement policy uses the benefits of both LFU and LRU policies. FBR divides the cache into three segments: a new segment, a middle segment, and the old segment. Data units are placed into sections based on their recency of usage. When a hit occurs, the hit counter is increased only for data units in the middle and old segments. When a replacement is needed, the policy chooses the data unit from the old segment with the smallest hit count. Author recommends setting on old section to 30% of cache size, and a new section to 60% of cache size [**68**].

LRFU

LRFU replacement policy employs both LRU and LFU replacement policies at the same time. LRFU calculates the so-called CRF (Combined Recency and Frequency) value for each data unit. This value quantifies the likelihood that the unit will be referenced in the near future. LFRU is suitable for use and was tested in database systems [**69**].

LRU-K

LRU-K replacement policy keeps the timestamps of the last K accesses to the data unit. When the cache is full, LRU-K counts so-called Backward K-Distance which leads the marked data unit to replace. The LRU-K algorithm is used in data base systems [**70**]. An example of LRU-K is **LRU-2**, which remembers the last two access timestamps for each data unit. It then replaces the data unit with the least recent

penultimate reference. Author recommend to set number of LRU queues to 3, and correlated reference period was to 7 [**71**].

LRD

LRD replacement policy replaces the data unit with the lowest reference density. Reference density is a reference frequency for a given reference interval. LRD has two variants of use. The first variant uses a reference interval which corresponds to the age of a page. The second variant uses constant interval time. Author recommends setting the interval for decreasing reference to 20, and the divisor of reference count to 1.8 [**72**].

ARC

ARC is similar to the 2Q replacement policy. The ARC algorithm dynamically balances recency and frequency. It uses two LRU-queues. These queues maintain the entries of recently evicted data units [**73**]. ARC has low computational overhead while performing well across varied workloads [**65**], [**74**]. ARC requires units with the same size; thus it is not suitable for caching whole files.

CRASH

CRASH is a low miss penalty replacement policy. It was developed for caching data blocks during reading data blocks from the hard disk. CRASH puts data blocks with contiguous disk addresses into the same set. When replacement is needed, CRASH chooses the largest set and replaces the block with the minimum disk address [**73**]. CRASH works with data blocks with the same size; thus CRASH is not suitable for caching blocks with different sizes.

Observing evaluation results of caching policies, presented policies usually achieve 5-10% improvement in read hit ratio indicator in comparison to LRU or LFU (e.g. 2Q algorithm gives approximately 5% improvement in the hit ratio over LRU [**67**]).

4.2 Caching Units

Before we introduce new caching policies, we have to decide what will be stored in the cache. The first option is to keep the data blocks with the same size in the cache;

the second option is to keep whole files in the cache. Keeping the blocks with the same size brings disadvantage in maintaining metadata for each stored block. Storing whole files brings disadvantage while requiring only a part of the file but the application has to download the whole file into the cache. Whole file caching is e.g. used in web content caching [**63**].

For making a decision, we monitored local Andrew File System (*/afs/kiv.zcu.cz/kiv/*) cell for a month. From gathered statistics, we observed that the most accessed files (over 98%) have the size from interval (0-5MB]. During the monitored period of time, users requested nearly 930,000 files. Number of requests to files with different file size is depicted in Figure 4.2. Because most requests require files stored in a few blocks, we decided to use the whole file as a basic unit, not having many units in the cache.

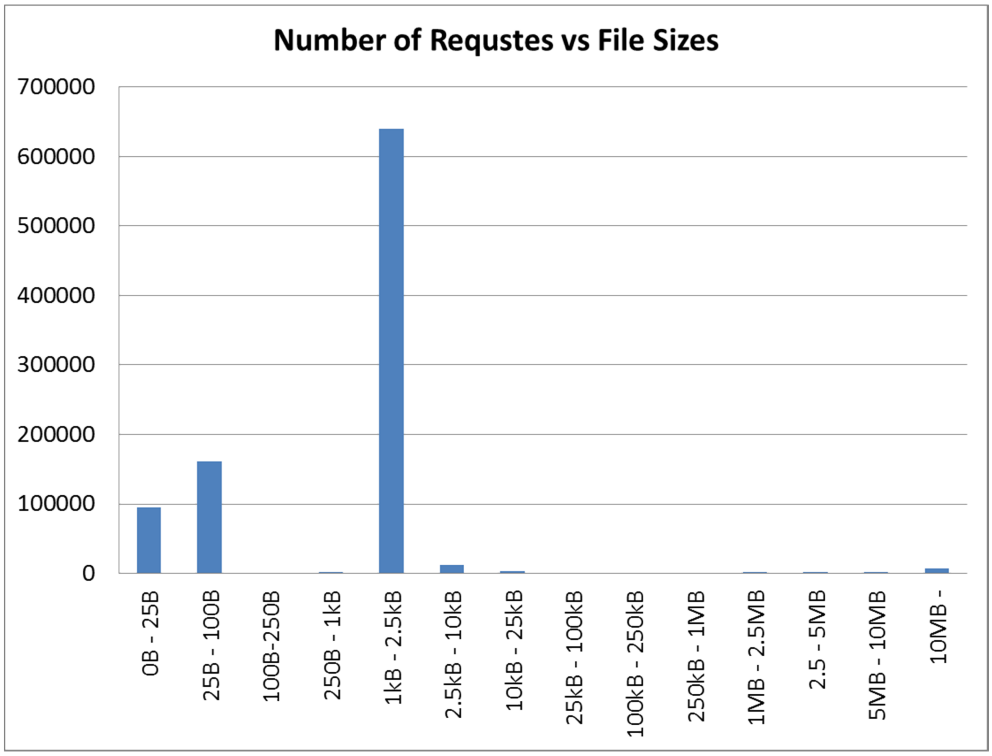

Figure 4.2: Number of Requests vs. File Sizes

4.3 The LFU-SS and LRFU-SS Algorithms

All caching algorithms mentioned were designed mainly for low-level I/O operations. These algorithms usually work with data blocks that have the same size. When replacement occurs, all the statistics-based and hybrid caching policies

mentioned choose the block to be removed from the cache based on statistics gathered during user requests. Moreover, all the caching policies have to store statistical information for all data blocks in the cache.

Considering caching policies also suitable for use in mobile devices, our first goal is to minimize costs of counting the priority of data units in the cache. This goal was set because mobile device are not as powerful as personal computers and their computational capacity is limited. The speed of data transfer from a remote server to the mobile device can vary caused by user movement. The amount of data transferred through cellular network is also limited by FUP. Thus, our second goal is to increase the cache hit ratio, and thereby decrease the network traffic caused by data transfer.

We present an innovated LFU algorithm called Least Frequently Used with Server Statistics (LFU-SS), and a hybrid algorithm called Least Recently and Frequently Used with Server Statistics (LRFU-SS) [**75**], [**76**].

4.3.1 LFU-SS

In LFU-SS, we use server and client statistics for replacement decisions. We consider server statistics first. Recall section 3.6, the database module of the server maintains metadata for the files stored in the DFS. These statistics are *file.readHits*, *file.writeHits* and *file.version*. Additionally, the Database Module can calculate *GlobalHits*. *GlobalHits* is a summation of the *file.readHits* of all files stored on the server. Calculation of the *GlobalHits* counter is a time-consuming operation. If we presume that the DFS stores thousands of files which are accessed by users, the value of variable *GlobalHits* is then much greater than the value of variable *file.readHits*, and we do not need to get the value of *GlobalHits* for each file access. The value of the *GlobalHits* counter is computed periodically, thus saving server workload.

The caching unit in our approach is the whole file. By caching whole files, we do not need to store read or write hits for each block of the file; we store these statistics for the whole file. Storing whole files also brings another advantage – calculation of priorities for replacement is not computationally demanding because

of the relatively low number of units in the cache compared to cache in which the data blocks with the same size are stored.

When the LFU-SS replacement policy must mark a file to be thrown out of the cache, LFU-SS works similarly to regular LFU. LFU-SS maintains metadata of files in a heap structure. In LFU-SS, we use a binary min-heap. The file for replacement is stored in the root node. When a user reads a cached file, the client read hits counter *file.client.readHits* is increased and the heap is reordered if necessary. The server statistics are only used for newly incoming files to the cache.

In a regular LFU policy, the read hits counter for a new file is initialized to one (the file has been read once). The idea of LFU-SS is that we firstly calculate the read hits counter from the statistics from the server. If the new file in the cache is frequently downloaded from the server, the file is then prioritized in comparison to a file which is not frequently read from the server. This leads to formula (1).

$$\text{file.client.readHits} = 1 + \text{GlobalHits}_{client} \cdot \frac{\text{file.readHits} - \text{file.writeHits}}{\text{GlobalHits}_{server}} \quad (1)$$

We first calculate the difference between read and write hits from the server. We prefer the files that have been read many times, and have not been updated so often. Moreover, we penalize the files that are often written and not often read. We do this in order to maintain the data consistency of the cached files. The variable $GlobalHits_{client}$ represents the summation of all read hits to the files in the cache. We add 1 because the user wants to read this file. We must store the read hits value as a decimal number for accuracy reasons when reordering files in the heap. The pseudo-code for LFU-SS is in Figure 4.3.

The disadvantage of using LFU-SS and general LFU relates to ageing files in the cache. If the file was accessed many times in the past, it still remains in the cache even if the file will not be accessed in the future again. We prevent this situation by dividing the variable *client.readHits* by 2. When the value of variable *client.readHits* reaches the threshold value, *client.readHits* variables of all cached files are divided by 2. The threshold value was set to 15 read hits experimentally.

```
Input: Request for file F
Initialization: heap of cached files records
              /*ordered by client's cache read hit counts */

if (F is not in cache)
{
      while (cache is full)
    {
          remove file with the least read hits;
          reorder heap to be min-heap;
    }
      compute initial F.client.readHits for file F;
      download file F into cache;
      insert metadata record to the heap;
      reorder heap to be min-heap;
}
else
{
      F.client.readHits++;
      reorder heap if necessary;
      upload client statistics to server;
      if (F.client.readHits > threshold)
      {
          for each FILE in cache do
                FILE.client.readHits = FILE.client.readHits / 2;
      }
}
```

Figure 4.3: Pseudo-code for LFU-SS

Using statistics from the server for gaining better results in the cache read hit ratio causes a need in updating these statistics. If the accessed files are provided from the cache, the statistics are updated only on the client side, and are not sent back to the server. In this case, the server does not provide correct metadata, and the policy does not work correctly. A Similar case occurs while using a cache on the server and client sides simultaneously [**22**]. To prevent this phenomenon, the client application periodically sends local statistics back to the server. The update message contains file ids and number of requests per each file since the last update. We show the experimental results for LFU-SS with and without uploading statistics to the server in sections 6.2, 6.4.1, 6.4.2, and 6.4.3.

We discuss the time complexity of using LFU-SS now. As mentioned before, we use a binary min-heap for storing metadata records. This heap is ordered the by read hits count. For cached files in LFU-SS, we use three operations: inserting a new file into the cache, removing a file from the cache, and updating file read hits:

- Operation inserting new file into cache has two steps: Insert file record into the heap with time complexity O(1), and reordering the heap structure with time complexity of O(logN). These time complexities are common for binary heap structures [**77**].
- Operation removing file record has time complexity of O(logN). We need to remove the record from the heap with the time complexity of O(1) and reorder the heap structure with complexity of O(logN).
- Operation updating file read hits has the time complexity of O(logN) in the worst case. The worst case occurs when the file is moved down from root to the leaf of the heap.

4.3.2 LRFU-SS

For mentioned hybrid caching replacement policies, the combination of LFU and LRU increases the cache hit ratio. Similarly, we use LFU-SS in combination with standard LRU. For the combination of these caching policies, we compute the priority of LFU-SS and LRU for each file in the cache. The priority of LFU-SS and LRU is from the interval [0, 65536), where a higher value represents a higher priority. The file with the lowest priority is replaced. For computing the final priority, we use the formula (2).

$$P_{final} = K_1 \cdot P_{LFU-SS} + K_2 \cdot P_{LRU} \qquad (2)$$

In computing the final priority, we can favour one of the caching policies by setting a higher value for K_1 or K_2 constants. The setting these constants is shown in section 6.2. Next, we focus on computing priority values $P_{LFU\text{-}SS}$ and P_{LRU} which were used in (2).

$P_{LFU\text{-}SS}$

The priority value for the LFU-SS algorithm is calculated by using linear interpolation between the highest and the lowest read hits values. For computing this priority, we use the formula (3).

$$P_{LFU-SS} = \frac{file.client.readHits - GlobalHits_{min,client}}{GlobalHits_{max,client} - GlobalHits_{min,client}} \cdot 65535 \qquad (3)$$

In formula (3), the values of variables $GlobalHits_{min,client}$ and $GlobalHits_{max,client}$ correspond to the highest and lowest read hits values. In the case that the file is new in the cache, we calculate read hits by using the formula from the previous section. We can expect that a new file in the cache is fresh and will also be used in the future. Despite computing read hits for a new file in the cache by using server statistics, new files in the cache still have a low read hits count. Therefore, we calculate the $P_{LFU\text{-}SS}$ for the new file in the cache in a different way. We use server statistics again and calculate the first $P_{LFU\text{-}SS}$ accordingly to formula (4).

$$P_{LFU-SS} = \frac{file.readHits}{GlobalHits_{server}} \cdot 65535 \qquad (4)$$

P_{LRU}

The least recently used policy usually stores the timestamp for last access to the file. If a replacement is needed, the file that has not been accessed for the longest time period is discarded. In our approach, we need to calculate the priority from the timestamp. We do this using formula (5).

$$P_{LRU} = \frac{T_{actual\ file} - T_{least\ recently\ accessed\ file}}{T_{most\ recently\ accessed\ file} - T_{least\ recently\ accessed\ file}} \cdot 65535 \qquad (5)$$

As shown in the formula, we again use linear interpolation for calculating P_{LRU}. We interpolate between $T_{least\_recently\_accessed\_file}$ and $T_{most\_recently\_accessed\_file}$. $T_{least\_recently\_accessed\_file}$ is the timestamp of the file that has not been accessed for the longest time period. $T_{most\_recently\_accessed\_file}$ is the timestamp of the file that has been accessed most recently.

The disadvantage of using LRFU-SS relates to computation priorities. We need to recalculate priorities for all cached units every time one cached unit is requested. We also need to reorder the heap of the cached files because of changes in these priorities. By caching whole files, we do not have many units in the cache, so these calculations are acceptable. The pseudo-code for the LRFU-SS is in Figure 4.4.

The LRFU-SS policy uses server statistics like LFU-SS. Using LRFU-SS causes the same problem with updating access statistics on the server side. We solved this

problem by periodically sending update messages back to the server. We show the experimental results for LRFU-SS with and without uploading statistics to the server in sections 6.2, 6.4.1, 6.4.2, and 6.4.3.

```
Input: Request for file F
Initialization: Min-Heap of cached files /*ordered by priority*/
              K1, K2 /*constants for computing Pfinal*/

if (F is not in cache)
{
    while (cache is full)
    {
          remove file with the least priority;
          reorder heap to be min-heap;
    }
    compute F.client.readHits;
    compute initial PLFU-SS for file F;
    compute PLRU for file F;
    compute Pfinal := K1 * PLFU-SS + K2 * PLRU;
    download and insert file F into cache;
    recalculate priorities of all files in the cache
          and simultaneously reorder the heap;
}
else
{
    F.client.readHits++;
    upload client statistics to server;
    if F.client.readHits > THRESHOLD
    {
          for each FILE in cache
                FILE.client.readHits = FILE.client.readHits / 2;
    }
    store new timestamp for file F;
    recalculate priorities of all files in the cache
          and reorder the heap;
}
```

Figure 4.4: Pseudo-code for LRFU-SS

As with LFU-SS, we discuss the time complexity of using LRFU-SS. Again, we use a binary min-heap for storing metadata records of cached files. We also employ three operations to the cached files: inserting a new file into the cache, removing a file from the cache, and accessing the file. Let N be the number of the cached files:

- The operation inserting a file entails recalculating time priorities of all cached files, which takes O(N) time. New priorities do not affect the heap structure because the recalculation maintains the min-heap property. After

recalculating new priorities, we insert a new file into the heap, which is O(logN). Then, insertion of a new file is O(N).

- The operation removing a file is O(logN) again.
- The operation accessing a file has the time complexity of O(N). As with inserting a new file, we need to recalculate priorities of all files taking O(N) time. For an accessed file, we need to recalculate the $P_{LFU\text{-}SS}$ priority and min-heapify the accessed file, which is O(logN). So, accessing a file takes is O(N) time.

5 Consistency Control

In previous section, we proposed caching policies suitable for mobile devices. If the cached objects remain in the cache for a long time period, they may be in inconsistent state if the original data are changed on the server side. For preventing this situation, so-called consistency control mechanisms were developed. However, these mechanisms were designed mainly for wired client; mobile devices were not taken into account.

In this section, we firstly introduce state-of-the-art in consistency control algorithms. Then, three novel algorithms are introduced. The first algorithm serves for client with reliable connection, the second and the third were developed mainly for mobile devices.

5.1 State of the Art in Consistency Control Algorithms

Three main views are usually used for maintaining cached data consistency and the invalidation of inconsistent data. The first view covers the demanded consistency level. The second view comes from the position of the server, and the third view says how to transfer consistent data to the cache [**78**]. These views are depicted in Figure 5.1 and described below.

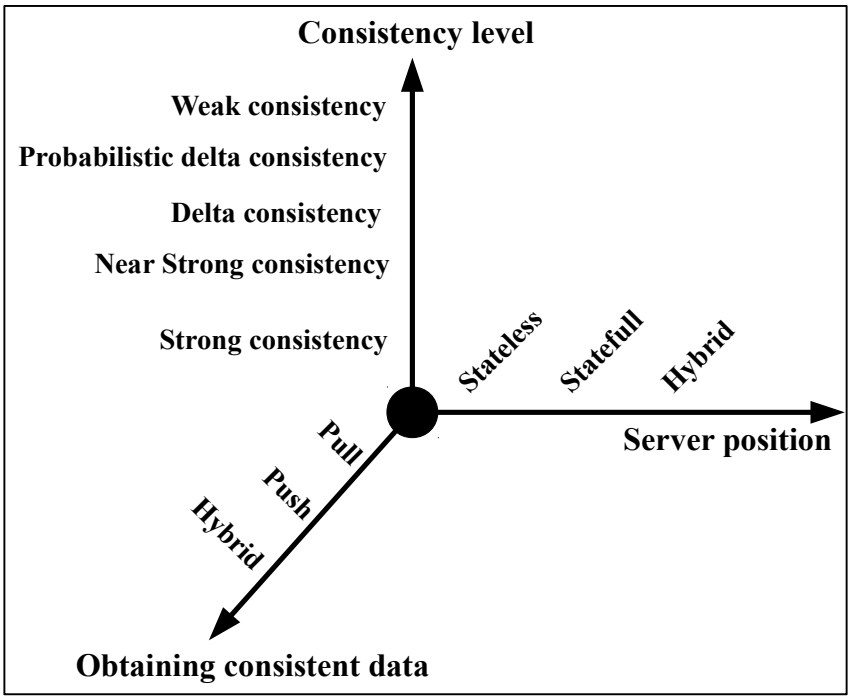

Figure 5.1: Maintaining cache consistency [**78**]

5.1.1 Obtaining Consistent Data

As soon as data are updated on the server and the client also has these data in the cache, the client application has to be informed about the change. Then the needed data should be transferred to the client cache.

By using a ***pull*** approach, the data are transferred to the clients' caches on demand [**20**]. The client application has to find out which data are in an inconsistent state. This strategy is also known as a client-initiated update. An example of pull-based strategy is a Time-To-Live algorithm. TTL is a time period when the data are considered to be in a consistent state. After expiration, the consistency of the cached data has to be examined. The TTL value can be constant (e.g. NFS for 30s [**14**]) or adaptive [**79**], [**80**].

Adaptive TTL usually provides a higher consistency level and also reduces network traffic [**79**]. Adaptive TTL can be calculated by multiplying a factor and the difference between the time in the cache and the timestamp of the last update [**81**] or by asking the user to set the mechanism [**82**]. Also the server history can be used for counting the TTL [**83**]. The client application can compare the time of the last modification and local time by open request [**45**].

The authors of [**84**] proposed using a hashtable to store the TTL values and the links to the cached objects. After expiration of the TTL value, the data can be renewed from the nearest cache node or from a data source. The newly updated data can be sent to the one-hop neighbours using a broadcast [**84**].

Opposite to the pull strategy, a ***push*** strategy means that the server sends the changed data immediately to the clients' caches. This strategy often uses call-back. The server has to store links to the clients who have the data in the cache. This strategy is also known as a server-initiated update. For mobile clients, this strategy is unusable. By using cellular networks, the provider usually uses NAT [**85**] because of depletion of available public IPv4 addresses. Thus, the server cannot notify the devices directly. At present, most of the mobile devices do not fully support IPv6 [**86**]; therefore IPv6 is not taken into account.

In an adaptive push strategy [**87**], there exist sets of nodes created automatically by using neighbour discovery. The cache update message is sent to the set of nodes, where the nodes re-send the message to another set of nodes. After flooding the nodes with the update message, the server with the updated data receives acknowledgement messages from all nodes [**87**].

A ***hybrid*** strategy usually supposes that all clients are on the same network segment. The server can use a broadcast message to inform all clients in the segment about newly updated data. A broadcast can also be used for sending updated data. All clients who have the inconsistent data in the cache simply listen and update their data. This strategy is often used in mobile ad-hoc networks (MANETs) [**79**].

The authors of [**35**] proposed using mobile agents to maintain cache consistency. In this concept, agents are used for obtaining shared locks for reading and exclusive locks for writing.

The authors of [**88**] proposed a hybrid mechanism which uses a combination of push and pull mechanisms. The server can send invalidation messages to all servers having inconsistent data in their cache. The client applications can also periodically renew the state of the cached data [**88**].

5.1.2 Server Position

A server position refers to an ability of the server to maintain the information about clients' caches.

A ***Stateless*** server does not keep any information about clients between particular sessions. For all clients using the cache, the server keeps and provides the timestamp of the last update of their cached data (e.g. NFS [**14**] or Cegor [**89**]). The client compares this time with the last update time of the cached data and decides which data are in an inconsistent state. In the stateless strategy, the clients usually obtain data using a pull algorithm. This strategy is commonly used when only the client can initiate the communication with the server.

A ***Stateful*** server keeps information about the data which are stored in clients' caches. After updating the data on the server side, the server uses call-back

to inform the client about inconsistency and also sends the updated data. This strategy is used e.g. in AFS [**14**], CODA [**68**], MAFS [**90**] or MSFSS [**36**]. The information about cached data is usually stored in non-persistent memory because this information is not considered to be significant for the server's functioning. A crash of the server causes loss of information about caches. It is also problematic to keep this information up-to-date because the mobile device can change its IP address caused by the user's movement. In this case, the server cannot inform the application about possible inconsistency. Changing of the IP address is caused by using different providers in different areas.

A ***Hybrid*** server usually does not keep information about data in clients' caches. A hybrid server usually uses a so-called Invalidation Report (IR). The IR is sent periodically to all clients in the network segment by using a broadcast. The IR contains information about newly updated data. After receiving the IR, the client can validate its local cache [**87**], [**91**].

The disadvantage of IR is in a long latency. This disadvantage can be reduced by using Updated IR (UIR). The UIR is a small fraction of the information related to the IR, and is replicated several times within the IR interval [**87**]. The IR messages can be collected at the gateways [**92**]. The gateway is then responsible for resending when the IR is lost. If the mobile client does not receive the IR message, the IR message can be demanded from the neighbouring client [**91**]. The clients can also verify their cached data within the IR interval when the cache hit occurs. This approach is called an Early Cache Validation Mechanism [**93**]. The authors of [**91**] use the Data Broadcast Cycle (DBR) to send the updated data to the clients. The DBR is within the IR interval. Additionally, the authors proposed algorithms to handle the loss of the IR or DBR messages by requesting the lost data from other mobile clients.

5.1.3 Demanded Consistency Level

The consistency level specifies a contract between a client and a system, wherein the system guarantees that the accessed data from the cache will be in a consistent

state [**94**]. There are several levels of consistency. Commonly used levels of consistency for cached data are strong, delta and weak.

In a ***strong*** consistency, all the requested data are in a consistent state at the time of the access. In other words, the cached data have to be the same as the data on the server side. This consistency can be achieved by using a stateful server and push approach.

In ***near strong*** consistency model, the file accessed from a cache is in a consistent state at the moment when the file is opened. This consistency can be achieved neither by using a stateful server and push approach nor by using a stateless server and pull approach [**79**].

A ***delta*** consistency guarantees that the data will be in a consistent state after a fixed time period δ since the data were updated on the server [**88**]. Delta consistency can be achieved by using reliable communication, the TTL or IR approach.

A ***weak*** consistency [**95**] guarantees that the data will be in a consistent state after a short time period. It usually uses an unreliable communication channel. Similarly to delta consistency, weak consistency uses the TTL or IR approach.

A ***probabilistic delta consistency*** is defined using two parameters: p (=probability) and δ (=time period) which have to be defined by a user. The consistency guarantees that for any node and any time, the probability that the cached data is stale within δ seconds is no less than probability p [**88**]. This level of consistency can be achieved by using a combination of the pull and push approaches.

5.2 Algorithms for Maintaining Cache Consistency

In this section, we describe novel algorithms for maintaining client-side cached data consistency for mobile clients. The first algorithm is suitable for users with reliable connection or for continuously connected users who use wired connection; the second and third one can be used mainly in mobile devices [**96**].

Firstly, we describe requirements for the server. KIV-DFS uses multi-master replication. The client application can access the data from any replica holder. In

the case that servers are statefull, it would be necessary to maintain the list of cached data for each client at each replica holder. Thus for efficiency reasons, we employ stateless data storage.

Mobile devices usually get an IPv4 private address from the internet provider. In this case, NAT has to be used at the gateway. So, the server cannot use call-back to contact the device. Thus, the application has to use a pull algorithm to obtain consistent data.

5.2.1 Consistency Control for Users with Reliable Connection

In this subsection, we provide a simple mechanism for maintaining near strong consistency.

The consistency control algorithm was adopted from the NFS. Instead of a timestamp of the last file update, we use the file.version counter (logical clock). During the file reading, a new version of the file may occur on the server-side. For maintaining this consistency, the file.version counter at the server side is compared with the local file.version counter which was initialized when the file was stored in the cache. If the file.versions do not match, the file is removed from the cache and the new copy is downloaded from the server. The new copy of the file may not have the same size as the removed file. In this case, the cache algorithm uses caching policy for replacement. The pseudo-code for this algorithm is depicted in Figure 5.2.

The advantages of this algorithm lie in the fact that it always provides the latest version of the file, and its implementation is simple. The disadvantage of this algorithm is the need for a permanent connection to the server. If the connection is lost and the user wishes to access the files, then the cache cannot guarantee the near strong consistency. For mobile clients using a wireless network, the permanent connection is also energy-intensive. If the user accesses frequently updated files, the consistency control overshadows the cache as performance improver. The consistency control may also cause a delay when the server is busy. Overall, this algorithm is suitable for clients with wired connection of for client with reliable mobile connection having permanent power supply.

```
Input: Request for file F
Initialization: Cache with cached files

If F not in cache
{
    If cache is full
    {
          DoReplacement();
    }
    DownloadFromServer(F);
    StoreInCache(F);
}
else {
    VersionFCache = Cache.getVersion(F);
    VersionFRemote = Server.getVersion(F);
    If (VersionFCache != VersionFRemote)
      {
          RemoveFromCache(F);
          If cache is full
          {
                DoReplacement();
          }
          DownloadFromServer(F);
          StoreInCache(F);
      }
}
ProvideFileFromCache(F);
```

Figure 5.2: Pseudo-code for near strong consistency

5.2.2 Consistency Control Mainly for Mobile Users (MMWP Algorithm)

Recalling the definition of consistencies, weak consistency guarantees that the cached files are in a consistent state within a short period of time after being updated. The client application does not need to be connected to the server during that period. Only after the period (or so-called Time-To-Live) expires, the application needs to be connected to the server to verify the versions of cached files. In mobile devices, this approach saves energy. The problem of this approach is how to set the TTL value for the cached files. When the TTL is too short, many verification messages have to be sent. On the other hand, when the TTL is too long, the files may be in an inconsistent state.

As mentioned above, the TTL can be set constantly or adaptively for all cached entities. We decided to set the TTL value adaptively to save network traffic and lower the load of the metadata server.

For setting the TTL values, we monitored local AFS cell (*/afs/kiv.zcu.cz/kiv/*) for a week from 27th January to 3rd February 2013 to obtain real data. The size of the log file is 203,200,942 bytes. It contains 135,195 requests for reading and 58,446 requests for writing. Each logged request contains a timestamp, user ID, IP address, file ID and the size of the file. The log does not contain information about the application which requested the file.

Examining the log file for write requirements between consecutive read requirements to a file, at most 5,466 inconsistencies could arise.

We examined the time periods between consecutive write requests to the files, i.e. the period when inconsistency cannot occur. From minimal writing periods of files with the same number of write requests, we counted the average minimal time period and the median value of the minimal periods. These times are depicted in Table 5.I.

Table 5.I: Time Periods of Writing New Files Content

Number of Writes	Number of Files	Average Minimal Period [ms]	Median of Minimal Periods [ms]
2	1691	1 031 082	128 000
3	30	525 226	62 500
4	21	2 464 190	51 000
5	5	2 882 500	43 000
6	13	1 769 429	49 500
7	7	400 125	33 400
8	19	30 200	31 000
9	9	56 222	22 000
10.15	72	80 958	30 000
16-20	22	78 217	30 000
21-30	44	31 089	17 000
31-50	16	9 706	1 000
51-100	48	11 483	1 000
101-500	47	2 041	1 000
501+	23	1 000	1 000

The average minimal period is influenced by the files which were updated at the beginning and at the end of the monitored time but not during the time period.

For setting TTL values, we will use the median of minimal periods, so cutting off the values from the ends of the time interval.

By adopting the TTL algorithm, we firstly find out the number of recent writes to the file. This information is provided by the server. Based on this information, we set the TTL period for each cached file to the median of minimal writing periods. In general, the TTL period could then be improved according to the occurrence of inconsistency, as will be presented later. We call this the MMWP adaptive TTL algorithm, MMWP in short. To our knowledge, the use of MMWP for computing the TTL value is novel. Pseudo-code for MMWP is depicted in Figure 5.3.

```
Input:
    Cached files
    Timer
    TTLVector /*TTL value for every cached file*/

    /*Find the least TTL value for timer*/
    Time leastTTL = findLeastValue(TTLVector);

    /*Set the least TTL value to the timer*/
    initiateTimer(Timer, leastTTL);

If Timer ticks
{
    For each file in cached files
    {
          If TTL of file expired
          {
                /*Evaluation of the file.version*/
                SendEvaluationMessageToServer(file.ID, file.version);
                Message reply = ReceiveMessageFromServer();

                /*Process file with new versions*/
                If (reply.version != file.version)
                {
                      RemoveFromCache(file);

                      /*New version of the file is larger*/
                      If cache is full
                            DoReplacement();
                      DownloadFromServer(file);
                      StoreInCache(file);
                }
                extendTTLValue(file, TTLVector);}
    }
}
```

Figure 5.3: Pseudo-code for MMWP adaptive TTL algorithm

This approach requires messages to be sent for each file and could therefore be improved by sending one verification message for a group of files. It can be expected that files with close numbers of write requests will have close medians of minimal writing periods and consequently also close TTL periods. The verifications messages for them could be grouped into one message and using one TTL period, thus saving network traffic.

Based on this idea, we divide the files in Table 5.I into groups. We form five groups of files with the distribution shown in Table 5.II. The number of groups was set on the basis of similarity of MMWP values. The file is assigned to one of the groups based on the file.writeHit counter. The file.writeHit counter is a part of the metadata which is provided to every file. The detailed setting of TTL values for each group is discussed in the section 6.5.2.

Table 5.II: Distribution of Files to the Groups

Number of Writes	Group	TTL
0-1	G_1	T_1
2-3	G_2	T_2
4-6	G_3	T_3
7-30	G_4	T_4
30+	G_5	T_5

We call this the MMWP batch adaptive TTL algorithm. The MMWP batch adaptive TTL algorithm is depicted in Figure 5.4.

Taking into account possible disconnection of the mobile clients, the MMWP adaptive TTL algorithm and the MMWP batch adaptive TTL fulfil weak consistency. These two algorithms are compared in the section 6.5.2.

```
Input:
    Groups of cached files
    Timer for each group
    Time Vector for Timers

If any timer ticks
{
    Group A = SelectCorrespondingGroup();
    Message Query;

    /*Preparation of the message*/
    For each file F in group A
    {
          Query = Query + ID(F) + Version(F);
    }
    SendMessageToServer(M);

    /*Response from the server*/
    Message reply = ReceiveMessageFromServer();
}

/*Process the files with new versions*/
For each file F in reply do
    {
          RemoveFromCache(F);

          /*New version of the file is larger*/
          If cache is full
                DoReplacement();
          DownloadFromServer(F);
          StoreInCache(F);
    }
}
```

Figure 5.4: Pseudo-code for MMWP batch adaptive TTL algorithm

6 Evaluation of Algorithms

In this section, we present an evaluation of proposed algorithms. We firstly show the evaluation of the caching policies using wired connection to the server. Then, a tool for comparison of the caching policies and cache consistency control called a Cache Simulator is presented. At the end of this section, results from this tool is presented and discussed.

6.1 Comparisons of Caching Policies and Consistency Control Algorithms

Before we provide results of the experiment, we focus on state-of-the-art in comparisons of caching policies and consistency control algorithms.

6.1.1 Commonly Used Comparisons of Caching Policies

The commonly used comparison of the policies is based on a cache hit ratio (see section 2.5.2) or on cache hit counts [**44**], [**22**], [**68**], [**97**], [**98**], [**99**]. A cache hit count is a number of requests which was served by a cache without the need of communication with data storage.

These statistics can be observed from trace-driven simulation. The trace can be collected by monitoring communication between the client applications and the server [**22**], [**68**], [**69**]. Often, the database [**71**], [**67**], [**69**] or web servers [**71**] are also used for collecting of traces. Except [**22**], none of the authors tells how is the trace collected. Froese and Bunt [**22**] changed the operating system kernel to record I/O events. By using the traces of real user behaviour, the simulation gives the most relevant results.

Another possibility of observing the trace is generating of user accesses randomly [**97**], [**98**], [**99**], [**45**], [**65**], [**71**]. Zipf or Normal distribution is used usually. The disadvantage of using a random generator is that the generator may not reflect the real user behaviour.

The basic caching unit used in many DFS is a data block. The data blocks have usually the same size. The blocks with the same size bring disadvantage in storing metadata for each block which are needed for caching policy. Moreover, the

policy has to store statistical information for each data block required to make a replacement decision. Then, choosing the block to be replaced is a time consuming operation because of high number of stored data blocks. The blocks with different sizes (files) are used in [**63**] where the experiments with storing large files in the cache were presented.

Since we use a whole file as a basic cached unit, we have to adopt other indicators which will serve for comparison of caching policies. These indicators are saved bytes which were presented in [**63**] and data transfer decrease which is new one. Saved bytes sums sizes of files which were found in the cache during simulation. Higher number in evaluation means better results. In opposite to saved bytes, we can compute data transfer decrease. This indicator sums the sizes of files which were not found in the cache and had to be downloaded from the server. In results, lower number of this indicator means better result for appropriate caching policy. Saved bytes and data transfer decrease indicators are complementary.

6.1.2 Indicators for Consistency Control Algorithms

None of the studies that are mentioned in this thesis presents a comparison of the consistency control algorithms. Thus, we had to adopt indicators for this comparison. We use different indicators for different algorithms [**96**].

For near strong consistency control, we monitor number of inconsistencies which are solved by this algorithm. Additionally, we monitor overall network traffic which is needed to download new versions of files and compute saved network traffic indicator. Saved network traffic indicator is difference between saved network traffic from caching policy and network traffic which is needed to consistency control.

For MMWP and MMWP batch, we monitor number of verification messages for maintaining consistency, size of files transferred for providing consistency and number of inconsistencies.

6.2 Caching Policies Evaluation Using a Wired Connection

For the first comparison of the novel caching polices, we used a wired connection to the server. We used the wired connection instead of wireless connection to

accelerate the experiments. The client application which communicates with the server is Total Commander [**100**] using plugin implementing KIVFS communication protocol. Because of high time consumption of the experiments, we implemented simple (RND, FIFO) and statistical (LFU, LRU) policies for comparison with LFU-SS and LRFU-SS policies. For a simulation of LRFU-SS, coefficients were set to K_1=0.35, K_2=1.1. We chose these coefficients after a series of experiments with LRFU-SS. For a simulation, we created 500 files with uniformly distributed random size between 1KB and 5MB on the server side. This distribution is based on analysis of the log file mentioned in section 4.2.

In a simulation scenario, we made 10,000 random requests on files where some of the files are prioritized and other files are accessed less often. The access rate was made by a random number generator and modulo function. We simulate LFU-SS and LRFU-SS with and without sending client statistics back to the server, to demonstrate the effect of sending client statistics.

Firstly, we observed the read hits count, then we computed read hit ratio. The experiment used cache sizes from 8MB to 512MB. These cache sizes were chosen because of limited capacity of mobile clients. Table 6.I summarizes cache read hit ratio for each of the implemented algorithms. For each simulated caching policy, we used the same scenario of accessed files.

The best algorithm in this scenario is LFU-SS. While using LFU-SS with cache capacities of 16MB and 32MB, we can achieve up to 11% improvement over commonly used LRU or LFU caching policies. When we use cache with larger capacity (64, 128, 256MB and 512MB), the improvement is up to 4% in cache hit ratio. The results also show the need of sending the local statistics back to the server to get better results.

On the other side, the cache read hits count or cache read hit ratio deals only with the count of the files in the cache that were found in the cache. We use whole file as a basic caching unit. Hence, the policy with the best read hits count does not have to be the best caching policy in saving data traffic because of variable file size.

Table 6.I: Cache Read Hit Ratio vs. Cache Size for Wired Client

Read Hit Ratio [%]/ Caching Policy	**Cache Size [MB]**						
Caching Policy	**8**	**16**	**32**	**64**	**128**	**256**	**512**
RND	2,96	5,71	10,31	16,09	25,78	40,45	62,43
FIFO	2,69	5,46	10,17	15,45	25,46	39,73	60,47
LFU	2,71	6,18	11,23	19,19	30,27	41,28	63,96
LRU	2,71	6,33	10,84	19,45	28,4	40,96	63,67
LFU-SS	6,56	13,15	21,73	25,35	31,65	42,31	64,68
LFU-SS without sending client statistics	2,53	5,53	11,05	18,45	28,67	35,39	55,78
LRFU-SS	4,5	10,34	15,39	23,73	30,8	42,12	64,43
LRFU-SS without sending client statistics	2,53	5,05	10,57	18,96	29,34	36,43	56,92

Secondly, we observed the saved data traffic while using various cache algorithms. The total size of transferred files was 22,1GB. We did the experiment with cache sizes from 8MB to 512MB again. Table 6.II summarizes the saved bytes indicator for different caching policies. From values in Table 6.II, we computed data transfer decrease indicators which are depicted in Table 6.III.

Table 6.II: Saved Bytes vs. Cache Size for Wired Client

Saved bytes [GB] / Caching Policy	**Cache Size [GB]**						
Caching Policy	**8**	**16**	**32**	**64**	**128**	**256**	**512**
RND	0,58	1,253	2,108	3,922	5,608	9,085	14,5415
FIFO	0,56	1,088	2,113	3,262	4,906	8,002	13,3953
LFU	0,42	1,029	1,959	4,492	5,755	8,228	14,3444
LRU	0,64	1,351	2,549	4,105	5,36	7,965	13,8128
LFU-SS	1,6	3,208	3,974	4,5	7,482	10,09	14,6886
LFU-SS without sending client statistics	0,61	1,62	2,61	5,15	6,87	8,97	13,55
LRFU-SS	0,85	2,391	3,731	6,372	8,362	10,87	15,4202
LRFU-SS without sending client statistics	0,56	1,283	2,585	4,291	6,401	8,516	13,2665

Table 6.III: Data Transfer Decrease vs. Cache Size for wired client

Data Transfer Decrease [GB] / Caching Policy	**Cache Size [GB]**						
Caching Policy	**8**	**16**	**32**	**64**	**128**	**256**	**512**
RND	21,52	20,85	19,99	18,18	16,49	13,01	7,56
FIFO	21,54	21,01	19,99	18,84	17,19	14,10	8,70
LFU	21,68	21,07	20,14	17,61	16,35	13,87	7,76
LRU	21,46	20,75	19,55	18,00	16,74	14,14	8,29
LFU-SS	20,50	18,89	18,13	17,60	14,62	12,01	7,41
LFU-SS without sending client statistics	21,49	20,48	19,49	16,95	15,23	13,13	8,55
LRFU-SS	21,25	19,71	18,37	15,73	13,74	11,23	6,68
LRFU-SS without sending client statistics	21,54	20,82	19,52	17,81	15,70	13,58	8,83

The best caching algorithm for cache sizes 8MB, 16MB, and 32MB is LFU-SS again. For larger cache capacity, the best caching policy is LRFU-SS with parameters K_1=0.35 and K_2=1.10. While using LRFU-SS with cache size of 265MB, we saved up to half of the network traffic. LFU-SS achieves up to 8% of improvement over LRU in small cache sizes. LRFU-SS achieves up to 10% of improvement over LRU and LFU in larger cache capacities.

6.3 CacheSimulator

Novel caching policies were compared only to small number of simple and statistical policies using wired client. The first results indicated good performance of novel policies compared to other ones. Therefore, we decided to compare novel policies to the variety of others policies under various parameter concerning requests' parameters and cache sizes. Except [**63**], all comparisons of caching policies assume that blocks with the same size are stored in the cache. However, in [**63**] no reusable tool for comparison is presented. For comparison of caching policies considering the whole file as a basic caching unit, we built a modular caching policies simulator, CacheSimulator in short. The CacheSimulator can also evaluate consistency control algorithms. In following text, we describe this tool.

The cache simulator was developed to overcome the main disadvantage of testing caching policies which lies in fact that testing is time-consuming operation. Additionally, the cache simulator provides the same conditions for every caching policy. The results from the cache simulator are not influenced by behaviour of different users. By using the cache simulator, we can compare the caching policies for various cache sizes without the need of deployment into real service repeatedly. The cache simulator was firstly introduced in [**101**].

Because a separate module implements particular caching policy, the simulator can be used for future comparison of new caching policies just adding a module with their implementation.

Although the most requested files are small, some of the demanded files can be larger than the cache capacity. In this case, the simulator does not store these files in the cache.

The cache simulator consists of three main parts: Server, Client, and Request generator.

A *Server* represents storage of files collection. Each file is represented by a unique ID (instead of file names) and size in bytes. Additionally, the server stores a number of read and writes requests for each file. All these metadata are provided for each file. Server can also compute global read hits. These metadata are provided on demand. Number of read and write hits and global read hits are used for replacement decision in LFU-SS and LRFU-SS caching policies. Number of write hits is also used for consistency control mechanism.

A *Client* is an entity which requests files from the server, uses the evaluated caching algorithm, and collects statistical information for caching algorithm. During the simulation, the client receives requests for file access from the Requests generator. For every request, client looks into a cache for the file. If the file is found, the counter for hits is increased by 1, and the counter of requested bytes and counter of saved byte are increased by the size of the file. Otherwise, only the counter of requested bytes is increased by the size of the file and the file is marked as cached. If the simulator evaluates consistency control, the client before each

request checks the consistency of cached files using corresponding algorithm. After checking the consistency, partial statistics are stored.

While evaluating consistency control, the simulator reads and processes read requests only from one pre-selected user; the write requests are processed from all users. Before the simulation starts, the files' statistics have to be initiated. For initialization, the log file is used twice. Supposing that the distribution of read and write requests is similar in a week before the log file was recorded, we can use it for initialization of files' statistics. Next, the cache was used and the consistency control was monitored for the selected user during simulation of access requests from the log file.

Requests generator is an entity which knows the files' ID on the server, and generates requests for these files. Requests generator can operate in two modes. In the first one, the generator reads log file, and produces requests accordingly. Events in log file contain ID and size of requested file, time when the file was requested, ID and IP address of the user who requested the file. For each user, the simulator creates and tracks all investigated caching policies separately for different cache sizes. Each event read from the log is assigned to the simulated user, and then the metadata of the file are stored in the user's caches. After the simulation, the simulator presents statistics for each user, each caching policy, and each cache size separately. In the second mode, the generator produces the requests randomly. For a simulation, we implemented normal (approximately) random generator, and random generator based on Zipf distribution. In a normal random generator, the user can set mean value and standard deviation. For a Zipf-like random generator, the user can set α parameter [**102**].

6.4 Caching Policies Evaluation Using Cache Simulator

In this section, we evaluate caching algorithms described in section 4.1. For testing, we ran three types of tests. The first test simulates user requests using normal random generator; the second test was run using Zipf random generator and the third test simulates user behaviour by reading an AFS log file.

For a simulation with random requests generators, we created 10,000 files on a server side. Both generators produced 100,000 requests to the files. This setting reflects our observation that number of requests should be ten times bigger than number of files. Files have uniformly distributed random size between 1KB and 5MB. This distribution is based on the observation from a log file again.

Numbers of requests to the files are not equal. Observing the log file, we see that some files were requested more often than the other files. We use the ordinal number of a generated file as a nominal one.

To simulate requests to the files in case of Normal distribution, because of its symmetry, the mean value for the normal random generator was set to the middle of the set of generated files (to the file which was generated in the 5000th place). The deviation was set to 100. It means that 68% of requests were to the files with ID's between 4900 and 5100. For the Zipf distribution, the value of α parameter was set to 0.75 based on the recommendation in [**102**].

A simulation from the AFS log creates files based on requests from the log. Used log file contains records from local AFS cell. We monitored local cell (*/afs/kiv.zcu.cz/kiv*) from 4th to 11th November 2012. The size of the log file is 821,732,953 bytes and contains 997,712 requests to the 178,704 different files. Number of users who requested files in this time period was 91.

We simulated the caching policies with sizes from 16MB to 512MB. Small cache sizes were observed because of usage in mobile devices. Higher cache sizes were simulated for usage in wired client applications.

As written above, we monitored cache hit ratio and saved network traffic (saved bytes) for every caching policy. We did not monitor data transfer decrease because data transfer decrease indicator and saved bytes indicator are complementary indicators. If one policy improves cache hit ratio and other policy improves saved bytes, we prefer the policy which improves saved bytes. For LFU-SS and LRFU-SS, we observed the statistics with and without sending the local statistics back to the server again.

Some of the caching policies require additionally settings for correct working. All setting respects the author's description of the algorithm.

6.4.1 Evaluation Using Normal Random Request Generator

Table 6.IV shows the cache hit ratio for a simulation where the requests were generated by a normal random requests generator. Table 6.V shows the results of saved bytes for the same simulation. Total size of transferred files without using any caching policy was 248.5GB. For rating caching policies, we select three best policies for every cache size. The best one is marked with red color, the second best has green color, and third best has blue color.

Table 6.IV: Read Hit Ratio for Normal Random Request Generator

Read Hit Ratio [%] / Caching Policy	**Cache Size [MB]**					
Caching Policy	**16**	**32**	**64**	**128**	**256**	**512**
2Q	2,28	4,70	9,78	19,53	**37,50**	**66,95**
Clock	1,70	3,49	7,06	14,03	27,88	53,15
FBR	**6,79**	**9,18**	**12,28**	**19,96**	35,52	65,28
FIFO	1,70	3,48	7,04	13,98	27,47	51,53
FIFO 2nd chance	1,70	3,50	7,16	14,30	28,60	55,43
LFU	**4,64**	6,05	11,11	18,08	32,23	60,80
LFU-SS	4,45	**8,21**	**12,97**	**21,45**	**37,50**	**68,10**
LFU-SS without sending client statistics	**5,88**	**8,67**	**12,92**	**20,13**	34,41	59,55
LIRS	2,11	4,09	7,94	15,67	30,28	58,14
LRDv1	1,70	3,49	7,14	14,32	28,78	57,34
LRDv2	1,70	3,50	7,10	14,11	28,02	53,79
LRFU	2,16	4,55	9,48	18,62	35,50	63,67
LRFU-SS	2,39	4,35	8,58	16,54	33,59	**68,46**
LRFU-SS without sending client statistics	2,16	3,95	9,40	18,82	**35,80**	62,61
LRU	1,70	3,49	7,09	14,14	28,16	54,18
LRU-K	1,72	4,16	9,02	18,02	**35,56**	66,26
MQ	1,83	3,71	7,45	14,79	29,06	55,35
MRU	1,30	2,20	3,90	7,67	15,54	30,90
RND	1,69	3,44	6,92	13,99	27,39	51,54

The results are also presented in graphic form using bar chart. On these charts, not all results are depicted. We decided to show results for novel caching policies LFU-SS and LRFU-SS (both with standard settings) and for other policies which achieved better results than the others (2Q, FBR, LFU, LRFU, LRU-K, LIRS and MQ). Cache hit ratio is depicted in Fig. 6.1 and saved bytes indicator is depicted in Fig. 6.2.

Table 6.V: Saved Bytes for Normal Random Request Generator

Saved Bytes [MB] / Caching Policy	**Cache Size [MB]**					
Caching Policy	**16**	**32**	**64**	**128**	**256**	**512**
2Q	4496	10140	21372	43912	86932	**163467**
Clock	4027	8441	17375	34813	69270	131716
FBR	2762	10203	20722	42138	83574	155873
FIFO	4030	8420	17297	34681	68095	127673
FIFO 2nd chance	4041	8484	17616	35579	71118	137355
LFU	**4967**	10428	19956	41307	81194	155508
LFU-SS	**5046**	**10640**	**22684**	**46679**	**93801**	**168539**
LFU-SS without sending client statistics	**5135**	**10886**	**23056**	**46033**	88053	153563
LIRS	4396	9313	18972	38383	74627	143790
LRDv1	4027	8462	17557	35566	71666	142241
LRDv2	4034	8454	17476	35038	69650	133251
LRFU	4454	9776	20945	42786	85863	157871
LRFU-SS	4833	9918	21601	44866	**90978**	**170002**
LRFU-SS without sending client statistics	4271	**10473**	**22391**	**45728**	**90989**	160489
LRU	4025	8442	17437	35110	70030	134151
LRU-K	4054	9363	19882	42489	85916	163330
MQ	4248	8941	18216	36629	72095	137025
MRU	3063	5315	9539	18795	38177	76776
RND	3958	8384	16966	34539	67765	127734
MRU	3063	5315	9539	18795	38177	76776
RND	3958	8384	16966	34539	67765	127734

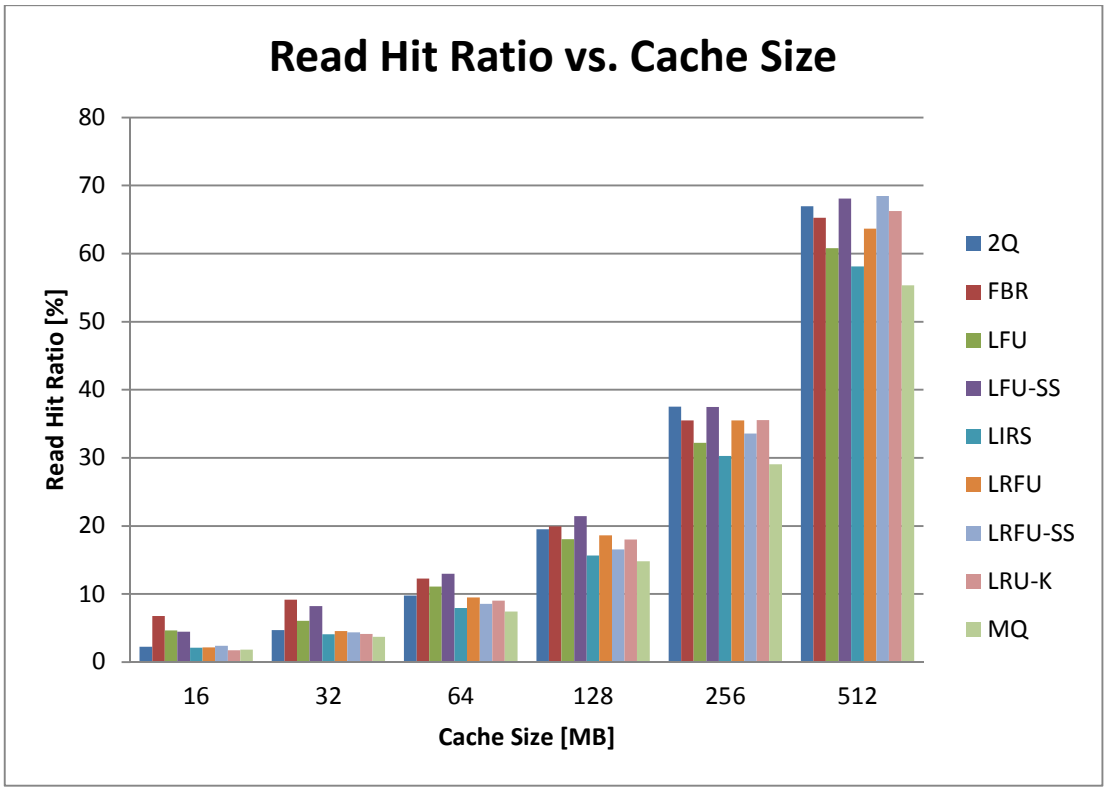

Figure 6.1: Read Hit Ratio for Normal Request Generator

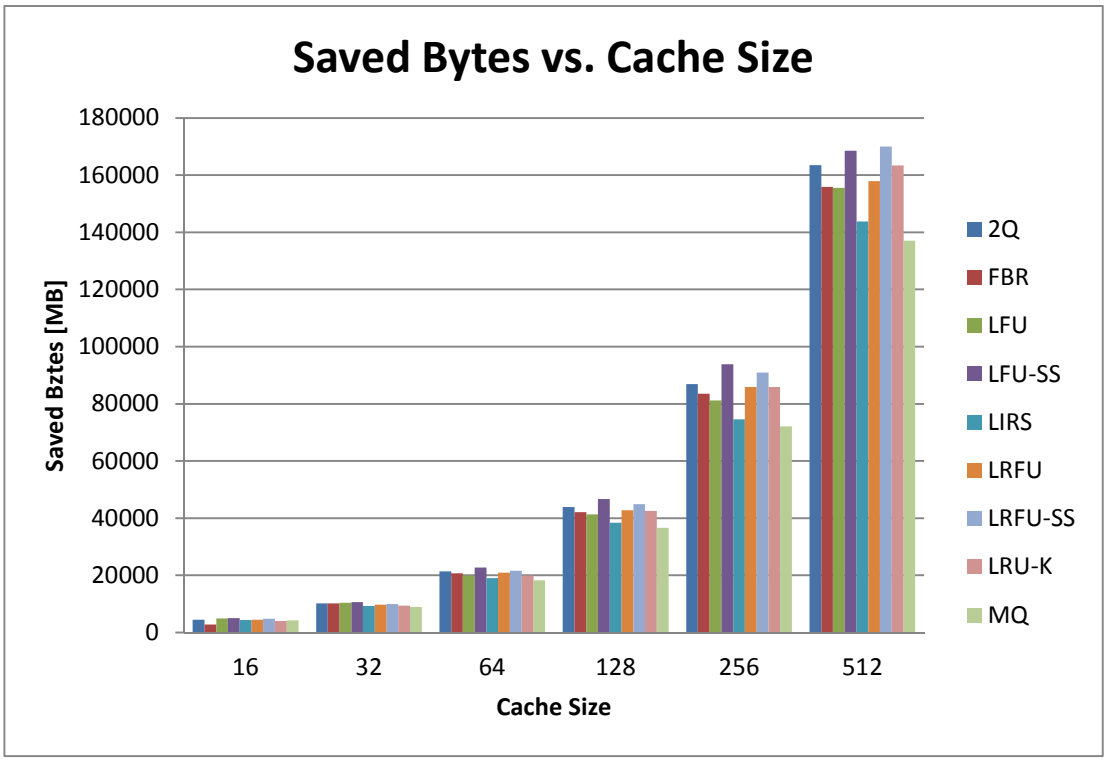

Figure 6.2: Saved Bytes for Normal Request Generator

As the results show, higher cache hit ratio does not mean that the caching policy is suitable for caching files in the DFS environment (see FBR policy). In the

cache hit ratio simulation, the best acting caching policy for caching files was LFU-SS for higher cache sizes, FBR for lower cache sizes. In saved bytes indicator, LFU-SS was the best one five times in six measurements. LRFU-SS is the second best policy. Sending local statistics back to the server influenced the results for LFU-FF, and for LRFU-SS. For lower cache sizes, they do not require to send local statistics to server. For higher cache sizes, it is necessary to send local statistics to server. Without sending local statistics, both policies work better for lower cache sizes, and worse for higher cache sizes.

6.4.2 Evaluation Using Zipf Random Request Generator

Next, we provide results from a simulation with Zipf random request generator. Table 6.VI shows cache hit ratio, and Table 6.VII shows saved bytes. Again, the results are also depicted using bar charts. Cache hit ratio is depicted in Fig. 6.3 and saved bytes in Fig. 6.4. Total size of transferred files without using a cache was 252.2GB. Similarly to previous simulation, we marked three best results with colours.

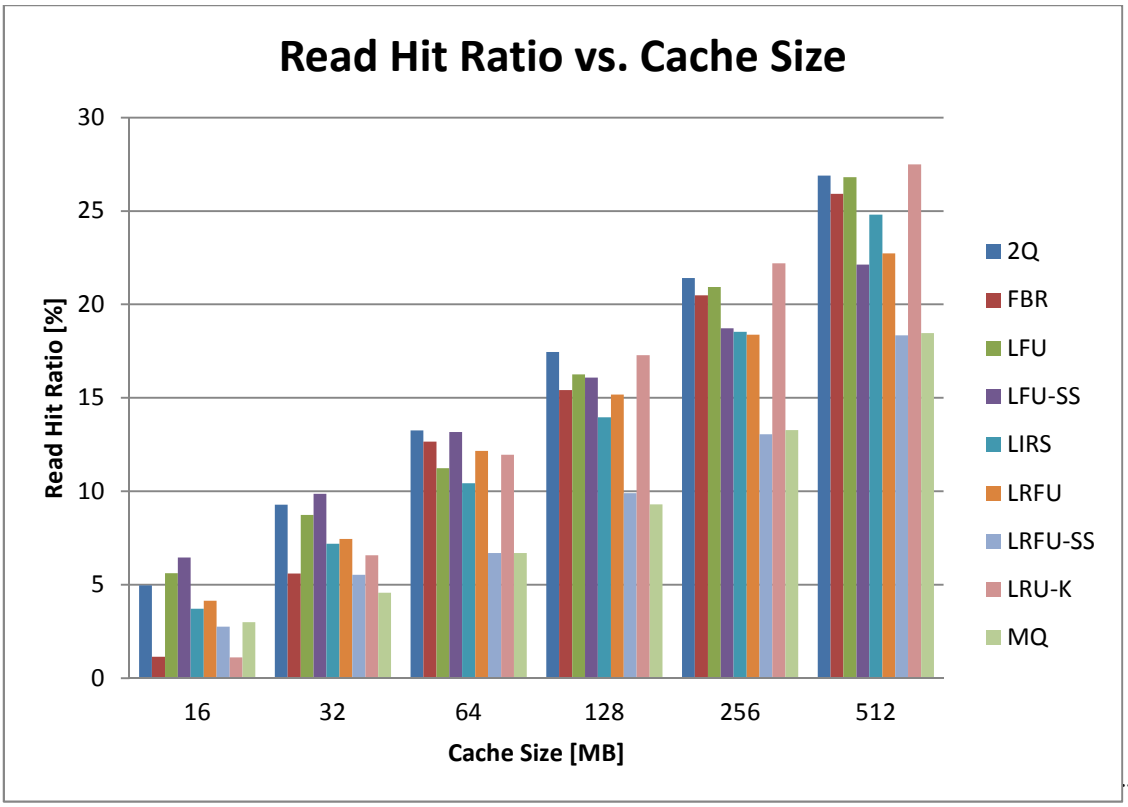

Figure 6.3: Cache Hit Ratio for Zipf Random Request Generator

In a simulation with Zipf random request generator, the results from cache hit ratio correspond with saved bytes indicator. In both simulations, the best acting caching policy is LFU-SS for lower cache sizes. For higher cache sizes, 2Q and LRU-K are better choice. Compared to previous simulation, LRFU-SS had worse results in both indicators. Again, sending local statistics back to the server influenced results for LFU-SS and LRFU-SS. LFU-SS achieved better results when it sent local statistics back to server. LRFU-SS did not need to send local statistics to server to gain better results.

Table 6.VI: Read Hit Ratio for Zipf Random Request generator

Read Hit Ratio [%] / Caching Policy	**Cache Size [MB]**					
Caching Policy	**16**	**32**	**64**	**128**	**256**	**512**
2Q	4,97	**9,29**	**13,27**	**17,46**	**21,42**	**26,89**
Clock	1,05	2,06	3,89	6,68	10,99	16,66
FBR	1,15	5,61	12,65	15,41	20,48	25,91
FIFO	1,05	2,02	3,74	6,27	10,03	15,14
FIFO 2nd chance	1,11	2,25	4,42	7,73	12,52	18,45
LFU	**5,62**	8,74	11,24	**16,26**	**20,93**	**26,80**
LFU-SS	**6,46**	**9,87**	**13,18**	16,09	18,72	22,13
LFU-SS without sending client statistics	3,56	4,29	5,64	8,12	12,66	18,20
LIRS	3,73	7,19	10,42	13,96	18,53	24,81
LRDv1	1,10	2,24	4,57	8,39	13,35	19,65
LRDv2	1,10	2,18	4,14	7,00	11,46	17,23
LRFU	4,15	7,45	12,17	15,18	18,38	22,74
LRFU-SS	2,76	5,54	6,70	9,90	13,06	18,34
LRFU-SS without sending client statistics	**6,45**	**9,42**	**12,76**	15,92	18,71	22,08
LRU	1,08	2,14	4,08	7,13	11,66	17,48
LRU-K	1,11	6,58	11,96	**17,28**	**22,20**	**27,50**
MQ	3,00	4,57	6,70	9,30	13,27	18,47
MRU	0,40	0,48	0,74	0,96	1,65	2,62
RND	1,04	2,03	3,73	6,25	10,04	15,11

Table 6.VII: Saved Bytes for Zipf Random Request Generator

Saved Bytes [MB] / Caching Policy	**Cache Size [MB]**					
Caching Policy	**16**	**32**	**64**	**128**	**256**	**512**
2Q	**14670**	**25530**	**36131**	**46584**	**57375**	**69782**
Clock	2911	5768	10959	18701	30378	45079
FBR	786	7880	31633	39933	53197	67961
FIFO	2899	5665	10491	17418	27537	40806
FIFO 2nd chance	3100	6395	12616	21880	34477	49726
LFU	**14281**	**24160**	31897	**44183**	**56385**	**68911**
LFU-SS	**18559**	**27415**	**36531**	43850	50477	59023
LFU-SS without sending client statistics	13619	22314	31896	40536	47919	56545
LIRS	10298	20573	28720	38108	49927	65436
LRDv1	3043	6323	13089	24085	36579	52889
LRDv2	3049	6161	11755	19710	31734	46544
LRFU	11899	20655	**34155**	41617	49569	60699
LRFU-SS	10162	18448	21046	28096	36004	49302
LRFU-SS without sending client statistics	3860	6132	10175	17509	30921	46222
LRU	3000	6011	11542	20088	32288	47239
LRU-K	3058	19458	33466	**46260**	**58578**	**70819**
MQ	9145	13890	19782	26440	36419	49743
MRU	974	1194	1790	2352	4112	6541
RND	2872	5744	10515	17419	27393	40712

6.4.3 Evaluation Using AFS Log File

Subsequent Tables 6.VII and 6.IX show results gained from a simulation using AFS log file. Table 6.VII shows cache hit ratio, and Table 6.IX shows saved bytes. Depicted results were gained from a user who made the most requests during a monitored time period. The user made 41,100 requests to files. Total size of transferred files without using a cache was 105,7GB. Similarly to previous results, we marked three best results with colours.

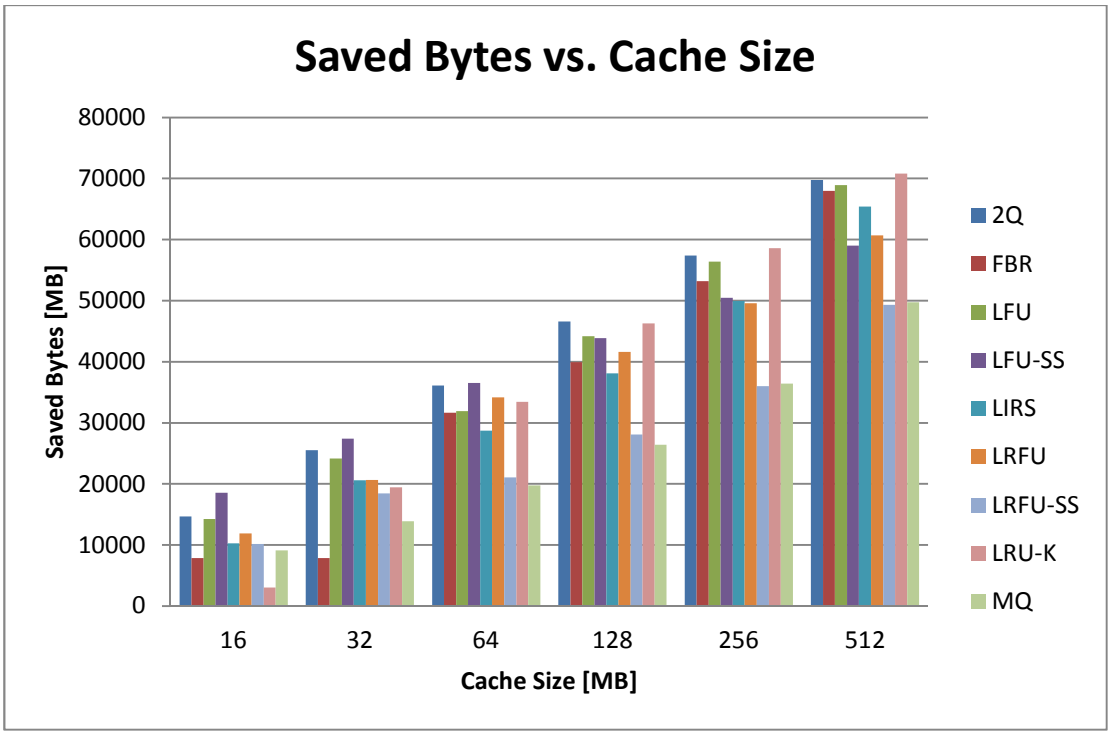

Figure 6.4: Saved Bytes for Zipf Random Request Generator

Again, the results are also depicted using bar charts. Cache hit ratio is depicted in Fig. 6.5 and saved bytes in Fig. 6.6.

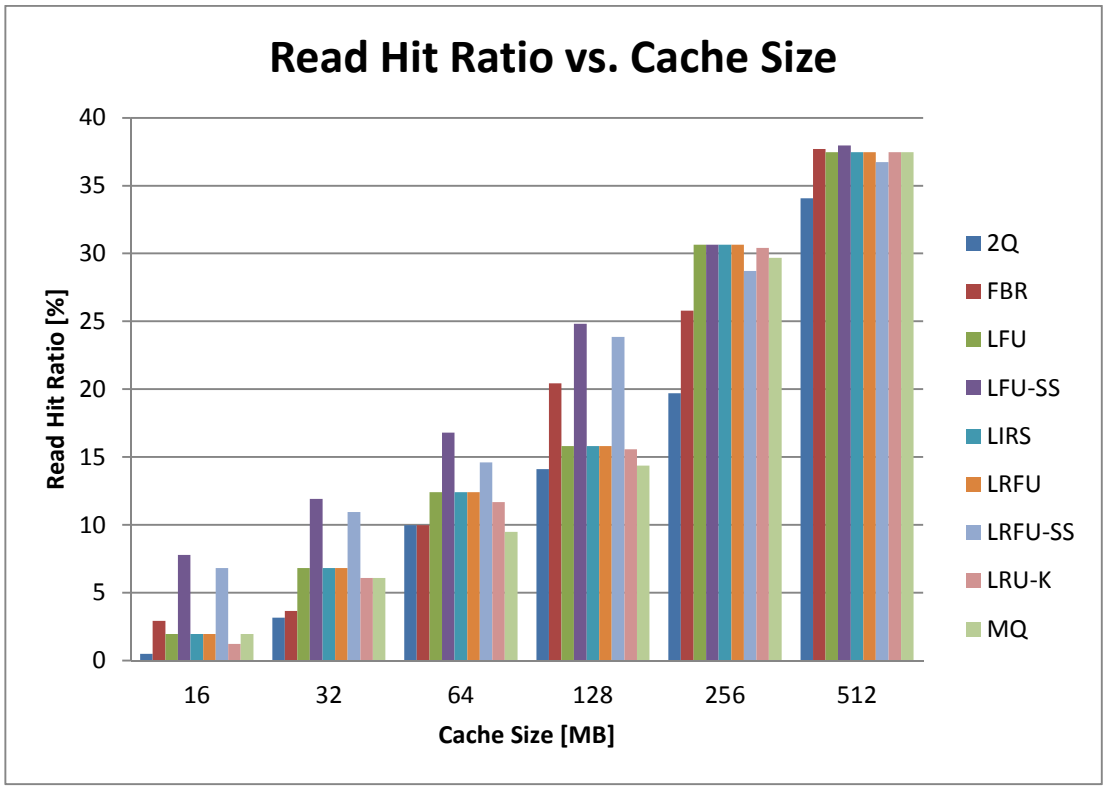

Figure 6.5: Cache Read Hit Ratio for Simulation from AFS log

The best acting caching policy in both indicators is LFU-SS. Results show that it is necessary to send local statistics to server for both LRFU-SS and LFU-SS to gain better results.

Table 6.VIII: Cache Read Hit Ratio for Simulation from AFS Log

Read Hit Ratio [%] / Caching Policy	**Cache Size [MB]**					
Caching Policy	**16**	**32**	**64**	**128**	**256**	**512**
2Q	0,49	3,16	9,98	14,11	19,71	34,06
Clock	1,22	2,92	5,84	9,98	23,36	36,01
FBR	2,92	3,65	9,98	20,44	25,79	**37,71**
FIFO	1,22	2,92	5,35	10,22	22,14	35,04
FIFO 2nd chance	1,22	3,16	6,33	11,19	27,49	37,47
LFU	1,95	6,81	12,41	15,82	**30,66**	37,47
LFU-SS	**7,79**	**11,92**	**16,79**	**24,82**	**30,66**	**37,96**
LFU-SS without sending client statistics	**4,38**	**10,95**	**16,06**	22,87	**29,68**	37,23
LIRS	1,95	6,81	12,41	15,82	**30,66**	37,47
LRDv1	1,22	3,16	6,57	11,92	28,47	37,47
LRDv2	1,22	3,16	6,33	10,71	24,82	35,77
LRFU	1,95	6,81	12,41	15,82	**30,66**	37,47
LRFU-SS	**6,81**	**10,95**	**14,60**	**23,84**	28,71	36,74
LRFU-SS without sending client statistics	4,13	**8,76**	13,87	**23,36**	28,22	36,98
LRU	1,22	2,92	6,33	10,95	25,30	36,74
LRU-K	1,22	6,08	11,68	15,57	**30,41**	37,47
MQ	1,95	6,08	9,49	14,36	**29,68**	37,47
MRU	1,22	2,68	9,49	20,92	26,28	36,74
RND	1,22	4,14	7,30	12,65	22,14	**37,52**

Table 6.IX: Saved Bytes for Simulation from AFS Log

Saved Bytes [MB] / Caching Policy	**Cache Size [MB]**					
Caching Policy	**16**	**32**	**64**	**128**	**256**	**512**
2Q	304	3542	11334	15382	21353	35825
Clock	1316	2834	5971	10626	24996	39772
FBR	1720	2631	8602	20948	26110	**40986**
FIFO	1316	2834	5262	10626	23984	38962
FIFO 2nd chance	1316	3238	6477	12549	29247	**40682**
LFU	2834	7792	13156	16496	**32586**	**40682**
LFU-SS	**5465**	**12448**	**18418**	**28032**	**33396**	**41492**
LFU-SS without sending client statistics	4149	9108	11031	21353	27122	35724
LIRS	2834	7792	13156	16496	**32586**	**40682**
LRDv1	1316	3238	6882	13460	30461	**40682**
LRDv2	1316	3238	6477	11840	25907	39569
LRFU	2834	7792	13156	16496	**32586**	**40682**
LRFU-SS	**5262**	**10322**	**16597**	**26717**	31271	40278
LRFU-SS without sending client statistics	**5161**	**10019**	**13662**	21050	28741	36027
LRU	1316	2834	6477	12043	27020	40176
LRU-K	1316	7286	12650	16293	**32485**	**40682**
MQ	2834	7388	10626	15180	31878	**40682**
MRU	1417	2834	9614	**21859**	28336	39772
RND	1113	3744	8197	13156	23782	**40986**

6.4.4 Summary of Caching Policies Evaluation

Overall, proposed algorithm LFU-SS has good results in all cases. If we take into account only simulation from the AFS log file, LFU-SS is the best caching policy in saved bytes indicator in comparison to other policies. It achieves up to 4% improvement in low cache sizes and 1% improvement in high cache sizes.

LRFU-SS achieves also good results over other caching policies. In simulations with random generators, only LFU-SS achieves good results; LRFU-SS does not.

Sending client statistics for both caching policies influences the results significantly.

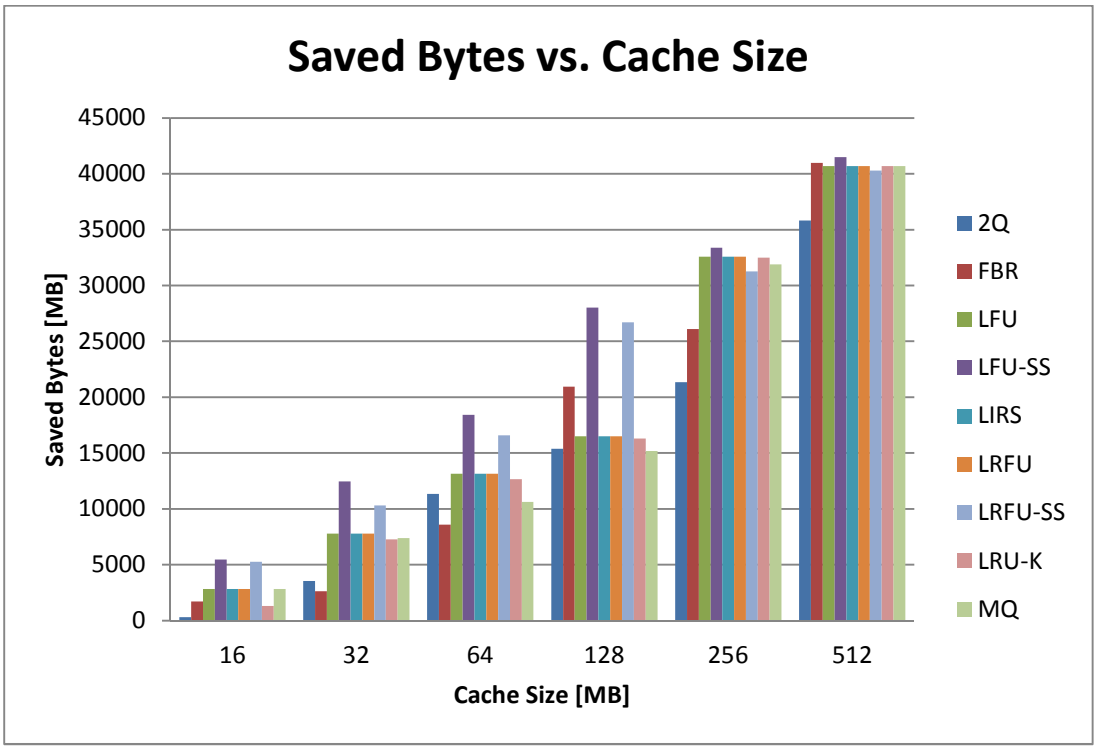

Figure 6.6: Saved Bytes for Simulation from AFS Log

6.5 Consistency Control Algorithms Evaluation

In this section, we provide evaluation of consistency control algorithms which were introduced in section 5.2. We used a simulation approach to evaluate the proposed consistency algorithms again. We use the log from AFS (see section 5.2.2) for generating read and write requests. From the log file, we choose the user with the highest ratio of accesses to updated files to accesses to files which were unchanged during the log period. For this user, the setting of TTL values for MMWP consistency control is the strictest in comparison to other users.

Chosen user made 9,509 read requests, from which 651 requests were to files which were updated on the server side. So, up to 651 inconsistencies can occur. This number is reduced if more write requests occur between consecutive read requests to the file, when these write requests cause one inconsistency.

The simulator reads and processes read requests from this user; the write requests are processed from all users. According to the proposed algorithms, when

the write request is processed, the file.version of the file is increased and file.writeHit counter is also increased. The file.version number and file.writeHit counter are equal until the file.writeHit counter is aged.

Before the simulation starts, the files' statistics have to be initiated. For initialization, the log file is used twice. Supposing that the distribution of read and write requests is similar in a week before the log file was recorded, we can use it for initialization of files' statistics. Next, the cache was used and the consistency control was monitored for the selected user during simulation of access requests from the log file. For all experiments, we used LFU-SS caching policy.

In the first experiment, we used various cache sizes from 32MB to 2048MB. Big cache size was chosen because more files are stored in the cache therefore more inconsistencies can occur. The total size of the requested files was 43.12MB. This would be the amount of data transferred when the cache is not used. In the experiment using the cache, we supposed that all requested files are immutable in order to exclude inconsistencies. We measured the saved network traffic, which is depicted in Table 6.X. The saved network traffic indicator shows the amount of the data which was served by the cache. Generally, files are not immutable only and using consistency control will reduce the saved network traffic because of updating the cache.

Table 6.X: Saved Network Traffic for LFU-SS Policy

LFU-SS	**Cache Size [MB]**						
	32	**64**	**128**	**256**	**512**	**1024**	**2048**
Saved network traffic [MB]	13,50	12,95	12,68	12,24	17,54	21,95	26,40

6.5.1 Evaluation of Near Strong Consistency Control

In the next experiment, we used near strong consistency control. We monitored the number of inconsistencies which were solved by this algorithm. Additionally, we monitored the overall network traffic which was needed to download new versions of files and the computed saved network traffic indicator. The results are depicted

in Table 6.XI. For this simulation, we presumed that the user is continuously connected to the server.

Table 6.XI: Results for Near Strong Consistency Control

Near Strong Consistency Control	**Cache size [MB]**						
	32	**64**	**128**	**256**	**512**	**1024**	**2048**
Number of inconsistencies without control	1	1	1	1	2	629	630
Size of transferred files to maintain consistency [MB]	3,04	3,04	3,04	3,04	3,05	7,87	7,88
Saved network traffic [MB]	10,46	9,91	9,64	9,20	14,49	14,08	18,52

We can observe how the saved network traffic is reduced and how the number of inconsistencies grows with the cache capacity due to more files stored in the cache. The saved network traffic is still considerable. The highest number of inconsistencies was for cache sizes of 1024MB and 2048MB because many files stored in the cache were changed on the server side. This state was expected.

6.5.2 Evaluation of MMWP and MMWP Batch Consistency Controls

In the next experiment, we explored the MMWP batch adaptive TTL algorithm. As mentioned in description of MMWP (section 5.2), we need to set the TTL for groups of the files for this consistency control; therefore we used the largest cache size. These TTL values form a vector which corresponds to the last column in Table 5.I.

Settings of TTL Values for MMWP Batch

For settings TTL for MMWP batch algorithm, we performed three scenarios in which different TTL were used. Main goal of these simulations was to set TTL for MMWP batch to produce no inconsistencies. The consistency of the accessed files in the simulation is verified when the file is requested by a user. We also presume that the client application can establish a connection to the server at any time, and the user demands daylong connectivity. For each time vector, we monitored the number of sent messages, the size of the files transferred for providing consistency and the number of inconsistencies for each group. The results of the experiment are depicted in Table 6.XII.

In the first scenario, we used the TTL according to the observation from Table 5.I (T_1=300s, T_2=100s, T_3=50s, T_4=30s, T_5=3s). The value of T_5 was set according to the lowest time for NFS [**103**], because the time period 1s from Table 5.I is too short. The time of T_1 was set manually based on previous observation on NFS [**103**]. Clearly, files in group G_1 were not in an inconsistent state. This scenario produced 251 inconsistencies in G_2, G_3 and G_4.

Table 6.XII: Results for MMWP batch adaptive TTL Consistency Control

MMWP Batch Adaptive Consistency Control		**Time Vector (T_1, T_2, T_3, T_4, T_5) [s]**		
		(300, 100, 50, 30,3)	**(300, 90, 25, 20, 15)**	**(300, 90, 20, 20, 10)**
Number of validation messages	G_1	1,440	1,440	1,140
	G_2	4,321	4,801	4,801
	G_3	8,642	17,285	21,606
	G_4	14,404	21,606	24,606
	G_5	144,045	28,809	43,213
Number of inconsistencies	G_1	0	0	0
	G_2	1	0	0
	G_3	249	249	0
	G_4	1	0	0
	G_5	0	3	0
Total number of messages		173,103	73,941	95,366
Number of updates		5,582	5,582	5,582
Size of transferred files to maintain consistency [MB]		18	18	18
Saved network traffic [MB]		8,69	8,79	8,89

In the second scenario, we reduced the times for groups G_2-G_4 (T_2=90s, T_3=25s and T_4=20s) to attempt to get a consistent state and increase the time for group G_5 (T_5=15s) to reduce the number of validation messages. The results of this scenario showed inconsistencies in groups G_3 and G_5.

From the results of the second scenario, we set the last scenario as follows. The times for G_5 and G_3 were reduced to attempt to get a consistent state again. The

new TTL values were T_5=10s and T_3=20s. This scenario produced no inconsistencies.

The total amount of sent messages is the lowest for the second scenario where inconsistent states occurred. Thus, the number of validation messages has to be higher, as the results of the third scenario showed. The number of updates and the size of transferred files show that this consistency control is more demanding on network traffic in comparison to near strong consistency control. The advantage of this control is that the files are available immediately when the user requests them. The new versions of the files are downloaded in the background. The saved network traffic indicator shows that MMWP batch adaptive TTL requires more network traffic in comparison to near strong consistency control.

Evaluation of MMWP and MMWP Batch

After we set the TTL for MMWP batch, we observed the number of verification messages for the MMWP adaptive TTL algorithm where the TTL value is assigned directly to the file, so the verification messages have to be sent for each file separately. In the experiment, we set TTL corresponding to the last scenario in the previous experiment. We measured the number of verification messages for the same user as in the third experiment. For comparison to MMWP and MMWP batch, we implemented constant TTL consistency control. The TTL value for constant TTL was set to 10s, which was the highest value producing no inconsistency state. The results are depicted in Table 6.XIII.

Table 6.XIII: Results for Constant and Adaptive TTL for each File

Algorithm	Total number of verification messages
MMWP Batch Adaptive TTL	95,366
MMWP Adaptive TTL	7,612,835
Constant TTL (10s)	71,233,638

Both the MMWP adaptive TTL algorithm and the MMWP batch adaptive TTL consistency controls verified the same data; thus the queries to the metadata database are the same. If we compare the number of messages from the MMWP

batch adaptive TTL control with the MMWP adaptive TTL algorithm, the batch control lowers the network communication substantially.

The measurement of the consistency control algorithms presented in [**78**] was performed using a random requests generator and showed a reduction of the number of messages needed for files verification. The results of MMWP and MMWP batch algorithms were obtained from a log file which monitored a real environment. These results show more significant reduction of the number of messages in comparison to [**78**].

7 Conclusion and Future Work

In this thesis, we proposed caching policies called LFU-SS and LRFU-SS. Novelty of these algorithms is in using server statistics and local statistics for making a replacement decision.

Our goals in developing new caching policies were to decrease network traffic, and minimize the cost of counting the priority of the data unit in the cache. The comparison of caching policies proved that the introduced algorithms perform better in comparison to other caching policies. Therefore, the idea of using server statistics was right.

By using cache, problem with data consistency has to be solved. To prevent inconsistent state, we presented three algorithms for maintaining cache consistency.

The first algorithm was adopted from NFS and it guaranties near strong consistency. Algorithms MMWP and MMWP batch use TTL to provide weak consistency. The novelty of these algorithms is in settings of TTL values. The TTL value for a file is set according to median of minimal writing periods gained from a log file. The number of verification messages can be lowered by using batch verification.

The comparison of MMWP and MMWP batch to Constant TTL shows significant lowering of verification messages number. The idea of using median of minimal writing periods to set TTL values was right.

Due to the absence of a tool for evaluation of novel algorithms, a new one called CacheSimulator was proposed and implemented. This tool was used to evaluation of all proposed algorithms.

In KIV-DFS, the cache is currently used only in client application. In our future work, we will focus on possibilities of using the cache on server side. On the server side, so-called hierarchical cache can be adopted. We can use statistical information to place the files in cache which will be stored on storages with various speeds (e.g. often requested file should be placed in cache located on SSD hard disk which provides files faster than standard hard disk).

Appendix A: Author's Activities

Publications Related to the Doctoral Thesis (ordered by release date)

BŽOCH, P. and ŠAFAŘÍK, J.: ***State of the Art in Distributed File Systems: Increasing Performance.*** *In 2011 2nd Eastern European Regional Conference on the Engineering of Computer Based Systems ECBS-EERC 2011. Los Alamitos: IEEE, 2011. pp. 153-154. ISBN: 978-0-7695-4418-2*

BŽOCH, P. and ŠAFAŘÍK, J.: ***Security and Reliability of Distributed File Systems.*** *In IDAACS´2011, Proceedings, vol. 1. Piscataway: IEEE, 2011. pp. 764-769. ISBN: 978-1-4577-1425-2*

BŽOCH, P. and ŠAFAŘÍK, J.: ***Increasing Performance in Distributed File Systems****. In Proceedings of the Eleventh International Conference on Informatics, Vol. I. Košice: EQUILIBRIA, s.r.o., 2011. pp. 147-152. ISBN: 978-80-89284-94-8*

BŽOCH, P. and ŠAFAŘÍK, J.: ***Algorithms for increasing performance in distributed file systems.*** *Acta Electrotechnica & Informatica, 2012, volume 2, pp. 24-30. ISSN: 1335-8243*

BŽOCH, P., MATĚJKA, L., PEŠIČKA, L., ŠAFAŘÍK, J.: ***Towards Caching Algorithm Applicable to Mobile Clients.*** *In Proceedings of the Federated Conference on Computer Science and Information Systems. Los Alamitos: IEEE, 2012. pp. 607-614. ISBN: 978-83-60810-51-4*

BŽOCH, P., MATĚJKA, L., PEŠIČKA, L., ŠAFAŘÍK, J.: ***Design and Implementation of a Caching Algorithm Applicable to Mobile Clients.*** *Informatica, 2012, volume 36, number 4, pp. 369-378. ISSN: 0350-5596*

BŽOCH, P. and ŠAFAŘÍK, J.: ***Simulation of Client-side Caching Policies for Distributed File Systems****. In EUROCON 2013. Piscataway: IEEE, 2013. pp. 679-686. ISBN: 978-1-4673-2231-7*

BŽOCH, P. and ŠAFAŘÍK, J.: ***Maintaining Cache Consistency for Mobile Clients in Distributed File System****. In 2013 3rd Eastern European Regional Conference on the*

Engineering of Computer Based Systems ECBS-EERC 2013. Los Alamitos: IEEE, 2013. pp. 55-62. ISBN: 978-0-7695-5064-0

Authorized Software

Cache Simulator [online], 2013

http://www.kiv.zcu.cz/en/research/downloads/product-detail-en.html?produkt_id=112

Related Talks

BŽOCH, P: ***Cache and Cache Algorithms.*** ReliSA Working Group Seminar, University of West Bohemia, Pilsen, Czech Republic, 4. 3. 2013.

BŽOCH, P: ***Zvyšování výkonu a udržování konzistentnosti dat v mobilních zařízeních pro distribuované systémy souborů.*** Distributed Systems Research Group Seminar, University of West Bohemia, Pilsen, Czech Republic, 30. 5. 2014.

Teaching Activities

2012 – 2014: PPA1 – Computers and Programming 1 (Java basics)

2010 – 2014: PPA2 – Computers and Programming 2 (Data structures)

2010 – 2012: ZOS – Fundamentals of Operating Systems

2010 – 2013: Lecturer at summer computer camp (tylidi.zcu.cz)

Bibliography

[1] Andrew S. Tanenbaum and Maarten Van Steen, *Distributed Systems : Principles and Paradigms*. Upper Saddle River: Prentice Hall, 2002.

[2] (2002) Transparency in Distributed Systems. [Online]. http://crystal.uta.edu/~kumar/cse6306/papers/mantena.pdf

[3] Kingston Technology Corporation. (2012) microSD Cards | Kingston. [Online]. http://www.kingston.com/us/flash/microsd_cards#sdc10

[4] Adam Khan. (2010, December) An Introduction to LTE. [Online]. https://sites.google.com/site/lteencyclopedia/home

[5] Dr.S.S.Riaz Ahamed, "Review and Implications of Time Division Multiple Access Techniques for Wireless Environment Services and Applications," *Journal of Theoretical and Applied Information Technology* , vol. 4, no. 9, pp. 807-812, Septempber 2008.

[6] Pavel Bžoch and Jiří Šafařík, "Algorithms for increasing performance in distributed file systems," *Acta Electrotechnica & Informatica*, vol. 12, no. 2, pp. 24-30, September 2012.

[7] Pavel Bžoch and Jiří Šafařík, "Increasing Performance in Distributed File Systems," in *Proceedings of the Eleventh International Conference on Informatics, Vol. I*, Rožňava, Slovensko, 2011, pp. 147-152.

[8] Pavel Bžoch and Jiří Šafařík, "Security and reliability of distributed file systems," in *IDAACS´2011, Proceeedings, vol. 1*, Praha, 2011, pp. 764-769.

[9] Pavel Bžoch and Jiří Šafařík, "State of the art in distributed file systems: Increasing Performance," in *2nd Eastern European Regional Conference on the Engineering of Computer Based Systems ECBS-EERC 2011*, Bratislava, 2011, pp. 153-154.

[10] Philippas Tsigas, "AFS Report," Department of Computing Science, Chalmers

University of Technology, Göteborg, Sweden, Lecture 2010.

[11] John H. Howard, "An Overview of the Andrew File System," in *Proceedengs of the USENIX Winter Technical Conference*, Dallas TX, 1988.

[12] Luboš Matějka, "Distribuované souborové systémy (Distributed File Systems)," in *Počítačové architektury & diagnostika : Pracovní seminář pro studenty doktorského studia : Lázně Sedmihorky*, 2005, pp. 125-128.

[13] T.Y.C. Woo and S.S Lam, "Authentication for distributed systems," in *Computer , vol. 25, no. 1*, 1992, p. 39.

[14] Rainer Többicke, "Distributed File Systems: Focus on Andrew File System/Distributed File Service (AFS/DFS)," in *Mass Storage Systems, 1994. 'Towards Distributed Storage and Data Management Systems.' First International Symposium. Proceedings., Thirteenth IEEE Symposium on*, Annecy, France, 1994, pp. 23-26.

[15] Jan Steuer. (2008) RPC, NFS, Automount. [Online]. http://www.fi.muni.cz/~kas/p090/referaty/2008-jaro/st/nfs.html

[16] B. Liskov, R. Gruber, P. Johnson, and L. Shira, "A replicated Unix file system," in *Management of Replicated Data, 1990. Proceedings., Workshop on the*, Houston, 1990, p. 11.

[17] M. Tim Jones. (2008, Jun) Anatomy of Linux journaling file systems. [Online]. http://www.ibm.com/developerworks/library/l-journaling-filesystems/index.html

[18] Holger Brueckner, Benny Chuang, Tiemo Kieft, and Steven McCoy. (2004, Sep) Gentoo Linux OpenAFS Guide. [Online]. http://61.153.44.88/gentoo/resources/document-listing/openafs.html

[19] Mahadev Satyanarayanan, "Scalable, secure, and highly available distributed file access," in *Computer*, 1990, pp. 9 - 18, 20-21.

[20] M. van Steen and G. Pierre, "Replication for Performance: Case Studies," in

Lecture Notes in Computer Science, Volume 5959., 2010, pp. 73-89.

[21] Microsoft. (2011, Sep) DFSR Overview. [Online]. http://msdn.microsoft.com/en-us/library/windows/desktop/bb540025(v=vs.85).aspx

[22] K.W. Froese and R.B. Bunt, "The effect of client caching on file server workloads," in *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on*, Wailea, HI , USA, 1996, pp. 150-159.

[23] František Barančík, *Protokoly pro ověřování uživatele (Protocols for user authentication)*. ZČU, Plzeň, 2009.

[24] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," , Incline Village, NV, 2010, pp. 1-10.

[25] S. Abdalla, I. Ahmad, Ewe Hong Tat, Gim Aik Teh, and Yong Lee Kee, "Towards Achieving a Highly Available Distributed File System," in *Advanced Communication Technology, The 9th International Conference on*, Gangwon-Do, 2007, pp. 2056-2060.

[26] Xin Sun, Jun Zheng, Qiongxin Liu, and Yushu Liu, "Dynamic Data Replication Based on Access Cost in Distributed Systems," in *Computer Sciences and Convergence Information Technology, 2009. ICCIT '09. Fourth International Conference on*, Seoul, 2009, pp. 829-834.

[27] Xiaodong Li and Chang Liu, "Towards a Reliable and Efficient Distributed Storage System," in *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference on*, 2005, pp. 301c - 301c.

[28] Chien-Min Wang, Tse-Chen Yeh, and Guo-Fu Tseng, "Provision of Storage QoS in Distributed File Systems for Clouds," in *Parallel Processing (ICPP), 2012 41st International Conference on*, 2012, pp. 189-198.

[29] D. Peric, T. Bocek, F.V. Hecht, D. Hausheer, and B. Stiller, "The Design and Evaluation of a Distributed Reliable File System," in *Parallel and Distributed*

Computing, Applications and Technologies, 2009 International Conference on, Higashi Hiroshima, 2009, pp. 348-353.

[30] (2008) CloudStore. [Online]. http://kosmosfs.sourceforge.net/features.html

[31] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google file system," in *SOSP '03 Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2003, pp. 29-43.

[32] (2010) Introduction to Gluster Versions 3.0.x. [Online]. http://download.gluster.com/pub/gluster/documentation/Introduction_to_Gluster.pdf

[33] Andr´ e Oriani and Islene C. Garcia, "From Backup to Hot Standby: High Availability for HDFS," in *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, 2012, pp. 131-140.

[34] (2010) Apache ZooKeeper - home. [Online]. http://zookeeper.apache.org/

[35] Jun Lu, Bin Du, Yi Zhu, and DaiWei Li, "MADFS: The Mobile Agent-Based Distributed Network File System," in *Intelligent Systems, 2009. GCIS '09. WRI Global Congress on*, Xiamen, 2009, pp. 68-74.

[36] Lihua Yu, Gang Chen, Wei Wang, and Jinxiang Dong, "MSFSS: A Storage System for Mass Small Files," in *Computer Supported Cooperative Work in Design, 2007. CSCWD 2007. 11th International Conference on*, Melbourne, Australia, 2007, pp. 1087-1092.

[37] Lei Wang and Chen Yang, "TLDFS: A Distributed File System based on the Layered Structure," in *NPC '07 Proceedings of the 2007 IFIP International Conference on Network and Parallel Computing Workshops*, Dalian, China, 2007, pp. 727-732.

[38] Ananth Devulapalli and Pete Wyckoff, "File Creation Strategies in a Distributed Metadata File System," in *2007 IEEE International Parallel and Distributed Processing Symposium, 2007*, Long Beach, CA, USA, 2007, p. 105.

[39] P. Carns et al., "Small-file access in parallel file systems," in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, Rome, Italy, 2009, pp. 1-11.

[40] Hsueh-Yi Chung, Che-Wei Chang, Hung-Chang Hsiao, and Yu-Chang Chao, "The Load Rebalancing Problem in Distributed File Systems," in *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, 2012, pp. 117-125.

[41] Bin Cai, Changsheng Xie, and Guangxi Zhu, "EDRFS: An Effective Distributed Replication File System for Small-File and Data-Intensive Application," in *Communication Systems Software and Middleware, 2007. COMSWARE 2007. 2nd International Conference on*, Bangalore, 2007, pp. 1-7.

[42] Dan Feng, Fang Wang Quan Zhang, "Metadata Performance Optimization in Distributed File System ," in *Computer and Information Science (ICIS), 2012 IEEE/ACIS 11th International Conference on* , 2012, pp. 476-481.

[43] J. Stender, B. Kolbeck, M. Hö gqvist, and F. Hupfeld, "BabuDB: Fast and Efficient File System Metadata Storage," in *Storage Network Architecture and Parallel I/Os (SNAPI), 2010 International Workshop on*, Incline Village, NV, 2010, pp. 51-58.

[44] Benjamin Reed and Darrell D. E. Long, "Analysis of caching algorithms for distributed file systems," in *ACM SIGOPS Operating Systems Review, Volume 30 Issue 3*, New York, NY, USA, 1996, pp. 12-17.

[45] A. Ermolinskiy and R. Tewari, "C2Cfs: A Collective Caching Architecture for Distributed File Access," in *High Performance Computing and Communications, 2009. HPCC '09. 11th IEEE International Conference on*, Seoul, 2009, pp. 642-647.

[46] P.T. Joy and K.P. Jacob, "A key based cache replacement policy for cooperative caching in mobile ad hoc networks," in *Advance Computing Conference (IACC), 2013 IEEE 3rd International*, Ghaziabad, 2013, pp. 383-387.

[47] Lamprini Konsta and Stergios V. Anastasiadis, "Hades: Locality-aware Proxy Caching for Distributed File Systems," 2009.

[48] D. Modi, R.K. Agrawalla, and R. Moona, "TransCryptDFS: A secure distributed Encrypting File System," in *Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2010 International Congress on*, Moscow, 2010, pp. 187-194.

[49] Zhiqian Xu and Hai Jiang, "HASS: Highly Available, Scalable and Secure Distributed Data Storage Systems," in *2009 International Conference on Computational Science and Engineering*, Vancouver, Canada, 2009, pp. 772-780.

[50] S. Chakravarthy and C. Hota, "Secure resilient high performance file system for distributed systems," in *Computer and Communication Technology (ICCCT), 2010 International Conference on*, Allahabad, Uttar Pradesh, 2010, pp. 87-92.

[51] R. Pletka and C. Cachin, "Cryptographic Security for a High-Performance Distributed File System," in *Mass Storage Systems and Technologies, 2007. MSST 2007. 24th IEEE Conference on*, San Diego, CA, 2007, pp. 24-27.

[52] Jiang Bian and R. Seker, "JigDFS: A secure distributed file system," in *Computational Intelligence in Cyber Security, 2009. CICS '09. IEEE Symposium on*, Nashville, TN, 2009, pp. 76-82.

[53] A. Boukerche, R. Al-Shaikh, and B. Marleau, "Disconnection-resilient file system for mobile clients," in *Local Computer Networks, 2005. 30th Anniversary. The IEEE Conference on*, Sydney, 2005, pp. 614-621.

[54] Azzedine Boukerche and Raed Al-Shaikh, "Servers Reintegration in Disconnection-Resilient File Systems for Mobile Clients," in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, Columbus, 2006, pp. 114-120.

[55] N. Michalakis and D.N. Kalofonos, "Designing an NFS-based mobile distributed file system for ephemeral sharing in proximity networks," in

Applications and Services in Wireless Networks, 2004. ASWN 2004. 2004 4th Workshop on, 2005, pp. 225-231.

[56] L. Matějka, L. Pešička, and J. Šafařík, "Distributed file system with online multi-master replicas," in *2nd Eastern european regional conference on the Engineering of computer based systems*, Los Alamitos, 2011, pp. 13-17.

[57] Marshini Chetty, Richard Banks, A.J. Brush, Jonathan Donner, and Rebecca Grinter, "You're capped: understanding the effects of bandwidth caps on broadband use in the home," in *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, Austin, Texas, USA, 2012, pp. 3021-3030.

[58] L. A. Belady, R. A. Nelson, and G. S. Shedler, "An anomaly in space-time characteristics of certain programs running in a paging machine," *Commun. ACM*, vol. 12, no. 6, pp. 349-353, June 1969.

[59] Richard P. Draves, "Page Replacement and Reference Bit Emulation in Mach," in *In Proceedings of the Usenix Mach Symposium*, 1991, pp. 201-212.

[60] Ph.D. Steven W. Smith, "Digital Signal Processors - Circular Buffering," in *The Scientist and Engineer's Guide to Digital Signal Processing*. San Diego: California Technical Publishing, 1998, pp. 506-509.

[61] Song Jiang, Feng Chen, and Xiaodong Zhang, "CLOCK-Pro: an effective improvement of the CLOCK replacement," in *ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference*, Berkeley, 2005, pp. 35 - 35.

[62] R.L Mattson, J. Gecsei, D.R. Slutz, and I.L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78-117, 1970.

[63] B. Whitehead, Chung-Horng Lung, A. Tapela, and G. Sivarajah, "Experiments of Large File Caching and Comparisons of Caching Algorithms," in *Network Computing and Applications, 2008. NCA '08. Seventh IEEE International*

Symposium on, Cambridge, MA, 2008, pp. 244-248.

[64] Hong-Tai Chou and David J. DeWitt, "An evaluation of buffer management strategies for relational database systems," in *VLDB '85 Proceedings of the 11th international conference on Very Large Data Bases - Volume 11*, 1985, pp. 127-141.

[65] Theodore Johnson and Dennis Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *In VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, 1994, pp. 439-450.

[66] Song Jiang and Xiaodong Zhang, "LIRS: An Efficient Low Interreference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *Proceedings of the 2002 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, (SIIMETRICS'02)*, Marina Del Rey, 2002, pp. 31-42.

[67] Yuanyuan Zhou, James F. Philbin, and Kai Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in *In Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, 2001, pp. 91-104.

[68] A. Boukerche and R. Al-Shaikh, "Towards Building a Fault Tolerant and Conflict-Free Distributed File System for Mobile Clients," in *Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 02, AINA 2006.*, Washington, DC, USA, 2006, pp. 405-412.

[69] Donghee Lee et al., "LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies," in *Computers, IEEE Transactions on*, 2001, pp. 1352 -1361.

[70] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *SIGMOD '93 Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, New York, 1993, pp. 297-306.

[71] Nimrod Megiddo and Dharmendra S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *FAST '03 Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, 2003, pp. 115-130.

[72] Wolfgang Effelsberg and Theo Haerder, "Principles of database buffer management," in *Journal ACM Transactions on Database Systems (TODS) Volume 9 Issue 4, Dec. 1984*, New York, 1984, pp. 560 - 595.

[73] Nong Xiao, YingJie Zhao, Fang Liu, and ZhiGuang Chen, "Dual queues cache replacement algorithm based on sequentiality detection," in *SCIENCE CHINA INFORMATION SCIENCES, Volume 55, Number 1, Research paper*, 2011, pp. 191-199.

[74] Woojoong Lee, Sejin Park, Baegjae Sung, and Chanik Park, "Improving Adaptive Replacement Cache (ARC) by Reuse Distance," in *9th USENIX Conference on File and Storage Technologies (FAST'11)*, San Jose, 2011, pp. 1-2.

[75] Pavel Bžoch, Luboš Matějka, Ladislav Pešička, and Jiří Šafařík, "Towards Caching Algorithm Applicable to Mobile Clients," in *Federated Conference on Computer Science and Information Systems (FedCSIS), 2012*, Wroclaw, 2012.

[76] Pavel Bžoch, Luboš Matějka, Ladislav Pešička, and Jiří Šafařík, "Design and Implementation of a Caching Algorithm Applicable to Mobile Clients," *Informatica*, vol. 36, no. 4, pp. 369-378, December 2012.

[77] Thomas H Cormen, Charles E Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction To Algorithms*, 3rd ed.: MIT Press and McGraw-Hill, 2009.

[78] Po-Jen and Chiu, Yu-Shian Chuang, "Efficient cache invalidation schemes for mobile data accesses," *Information Sciences: an International Journal*, vol. 181, no. 22, pp. 5084-5101, November 2011.

[79] Kassem Fawaz and Hassan Artail, "DCIM: Distributed Cache Invalidation Method for Maintaining Cache Consistency in Wireless Mobile Networks," *IEEE Transactions on Mobile Computing*, vol. 12, no. 4, pp. 680-693, April 2013.

[80] Z. Wang, S.K. Das, Hao Che, and M Kumar, "A Scalable Asynchronous Cache Consistency Scheme (SACCS) for mobile environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 11, pp. 983-995, November 2004.

[81] Vincent Cate, "Alex - a Global Filesystem," in *IN PROCEEDINGS OF THE 1992 USENIX FILE SYSTEM WORKSHOP*, 1992, pp. 1-12.

[82] Jeong-Joon Lee, Kyu-Young Whang, Byung Suk Lee, and Ji-Woong Chang, "An Update-Risk Based Approach to TTL Estimation in Web Caching," in *Proceedings of the 3rd International Conference on Web Information Systems Engineering*, Washington, DC, USA, 2002, pp. 21-29.

[83] Laura Bright, Avigdor Gal, and Louiqa Raschid, "Adaptive pull-based policies for wide area data delivery," *ACM Trans. Database Syst.*, vol. 31, no. 2, pp. 631-671, June 2006.

[84] N.S. Fatima and P.S.A. Khader, "Enhanced Adaptive data cache invalidation approach for mobile ad hoc network," in *Electronics Computer Technology (ICECT), 2011 3rd International Conference on*, 2011, pp. 76-80.

[85] Charles M. Kozierok. (2005, September) IP Network Address Translation (NAT) Protocol. [Online]. http://www.tcpipguide.com/free/t_IPNetworkAddressTranslationNATProtocol.htm

[86] David Boreham. (2011, June) Mobile Device IPv6 Support Status. [Online]. http://wiki.nuevasync.com/wiki/bin/view/Public/deviceIpv6Support

[87] G. Anandharaj and R. Anitha, "A Distributed Cache Management Architecture for Mobile Computing Environments," in *IEEE International Advance Computing Conference, 2009. IACC 2009.* , 2009, pp. 642-648.

[88] Yu Huang et al., "Flexible Cache Consistency Maintenance over Wireless Ad Hoc Networks," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 8, pp. 1150-1161, August 2010.

[89] Weisong Shi, S Santhosh, and Hanping Lufei, "Cegor: an adaptive distributed file system for heterogeneous network environments," in *Proceedings of the Tenth International Conference on Parallel and Distributed Systems, 2004. ICPADS 2004.*, 2004, pp. 145-152.

[90] Benjamin Atkin and Kenneth P. Birman, "Network-Aware Adaptation Techniques for Mobile File Systems," in *Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, Washington, DC, USA, 2006, pp. 181-188.

[91] Thu Tran Minh Nguyen and Thuy Thi Bich Dong, "An adaptive cache consistency strategy in a disconnected mobile wireless network," in *2011 IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, 2011, pp. 256-260.

[92] Wenzheng Xu, Weigang Wu, Hejun Wu, Jiannong Cao, and Xiaola Lin, "CACC: A Cooperative Approach to Cache Consistency in WMNs," *IEEE Transactions on Computers*, no. PrePrints, pp. 1-14, 2013.

[93] Yeim-Kuan Chang, I-Wei Ting, and Tai-Hong Lin, "Dynamic Cache Invalidation Scheme in IR-based Wireless," in *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*, 2008, pp. 697-704.

[94] S. Qadeer, "Verifying sequential consistency on shared-memory multiprocessors by model checkin," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 14, no. 8, pp. 730-741, 2003.

[95] Sarita V. and Hill, Mark D. Adve, "Weak Ordering - A New Definition," *SIGARCH Comput. Archit. News*, vol. 18, no. 3a, pp. 2-14, June 1990.

[96] Pavel Bžoch and Jiří Šafařík, "Maintaining Cache Consistency for Mobile Clients in Distributed File System," in *2013 IEEE Third Eastern European Regional Conference on the Engineering of Computer Based Systems ECBS-EERC 2013*, Budapest, 2013, pp. 55-62.

[97] Qinglong Hu , Wang-chien Lee , Dik Lun Lee Jianliang Xu, "Performance Evaluation of an Optimal Cache Replacement Policy for Wireless Data Dissemination," *Journal IEEE Transactions on Knowledge and Data Engineering Volume 16 Issue 1*, pp. 125-139, January 2004.

[98] Hui Chen, Yang Xiao, and Xuemin (Sherman) Shen, "Update-Based Cache Access and Replacement in Wireless Data Access," *Journal IEEE Transactions on Mobile Computing Volume 5 Issue 12*, pp. 1734-1748 , December 2006.

[99] Ajey Kumar, Manoj Misra', and A. K. Sarje, "A new cache replacement policy for location dependent data in mobile environment," in *Wireless and Optical Communications Networks, 2006 IFIP International Conference on*, 2006.

[100] Total Commander - home page. (2012, november) Christian Ghisler. [Online]. http://www.ghisler.com/

[101] Pavel Bžoch and Jiří Šafařík, "Simulation of Client-side Caching Policies for Distributed File Systems," in *Eurocon 2013*, Zagreb, 2013, pp. 679-686.

[102] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: evidence and implications," in *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, New York, NY, 1999, pp. 126-134.

[103] Jehn-Ruey Jiang, "Consistency in NFS and AFS," National Central University, Department of Computer Science and Information Engineering, Jhongli City, Lecture notes 2008.